SECURE VIRTUAL ARCHITECTURE: SECURITY FOR COMMODITY SOFTWARE SYSTEMS

BY

JOHN T CRISWELL

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Vikram Adve, Chair and Director of Research
Associate Professor Madhusudan Parthasarathy
Associate Professor Sam King
Professor Greg Morrisett, Harvard University

Abstract

Commodity operating systems are entrusted with providing security to the applications we use everyday, and yet they suffer from the same security vulnerabilities as user-space applications: they are susceptible to memory safety attacks such as buffer overflows, and they can be tricked into dynamically loading malicious code. Worse yet, commodity operating system kernels are highly privileged; exploitation of the kernel results in compromise of all applications on the system.

This work describes the Secure Virtual Architecture (SVA): a compiler-based virtual machine placed between the software stack and the hardware that can enforce strong security policies on commodity application and operating system kernel code. This work describes how SVA abstracts hardware/software interactions and program state manipulation so that compiler instrumentation can be used to control these operations, and it shows how SVA can be used to protect both the operating system kernel and applications from attack. Specifically, this work shows how SVA can protect operating system kernels from memory safety attacks; it also shows how SVA prevents a compromised operating system kernel from adversely affecting the execution of trusted applications by providing application memory that the operating system kernel cannot read and write and secure application control flow that the operating system cannot corrupt.

Acknowledgements

I have so many people to thank that I'm likely to have forgotten someone. If I've forgotten you, please accept my apologies in advance.

First, I thank Rosa Rosas for encouraging me to finish my undergraduate degree and for all of her patience and support while I worked on this dissertation. Hopefully the many nights I spent working on my dissertation gave you time to work on your dissertation.

I thank my parents for nurturing my intellectual pursuits, for allowing me to spend late nights working on homework, and for their financial assistance in attending the University of Illinois. It looks like all that undergraduate tuition money was well spent.

I thank my old friend, Forest Godfrey, for kindling my love of computing. My whole career in computing is pretty much his fault. I also thank the rest of the Godfrey family, Eric, Ann Marie, and Brighten, for their friendship and for being my role models.

I thank my teachers for providing the foundation upon which my graduate education is built. While I owe them all, I extend special thanks to Marsha Woodberry, Christine Stewart, James Watson, Mike Troyer, and Joe Barcio of the Ripon School District and the late Michael Faiman of the University of Illinois.

I thank my colleagues at Argus Systems Group for kindling my love of computer security. I extend special thanks to Randy Sandone and Paul McNabb for giving me my first real job, Jason Alt and Mikel Matthews for their support, and Jeff Thompson for teaching me to always think critically.

I thank my advisor, Vikram Adve, for motivating me to always do better and for getting me to accomplish more than I ever thought possible. I also thank him for his confidence and mentorship over the years. The greatest obstacle I faced was having confidence in myself, and you've helped me overcome that. Thank you.

I thank my committee members Madhusudan Parthasarathy, Sam King, and Greg Morrisett for their interest in and feedback on my research work. Their suggestions have made this work better.

I thank my collaborators and co-authors, including Nathan Dautenhahn, Nicolas Geoffray, Dinakar Dhurjati, Brent Monroe, Andrew Lenharth, and Swarup Sahoo. I also thank my fellow graduate students Will

Dietz, Arushi Aggarwal, and Rob Bocchino. I thank Pierre Salverda, David Raila, and Roy Campbell for numerous insightful discussions about the design of SVA-OS.

I thank all of our shepherds and anonymous reviewers for their extensive and helpful feedback on my papers which contributed to this dissertation.

I thank the LLVM community for providing a great compiler infrastructure with which to work. In particular, I thank Chris Lattner for his drive and hard work which made LLVM possible. I also thank Bin Zeng, Gang Tan, and Greg Morrisett for sharing their x86 CFI instrumentation pass with me. I also thank the FreeBSD community for providing a commodity operating system that compiles with LLVM/Clang.

Finally, I thank Phil Wall for all the good times we had playing board games and drinking fine beer. Phil, you were more important to us than you ever knew. Rest in peace, kindred spirit and friend.

Dedicated to Rosa Rosas.

Table of Contents

List of Tables . . . . . **xi**

List of Figures . . . . . **xii**

Chapter 1 Introduction **1**
1.1 Motivation 1
1.2 Secure Virtual Architecture 2
1.3 Enforcing Security Policies with SVA 3
1.4 Contributions 4
1.5 Organization 5

Chapter 2 Virtual Architecture Support for Operating System Kernels **6**
2.1 Introduction 6
2.2 Background: VISC Architectures and LLVA 8
2.3 Design of the OS Interface 9
2.3.1 Design Goals for the OS Interface 9
2.3.2 Structure of the Interface 10
2.3.3 Virtual and Native System State 10
2.3.4 Manipulating Virtual and Native State 11
2.3.5 Interrupts and Traps 12
2.3.6 System Calls 13
2.3.7 Recovery from Hardware Faults 14
2.3.8 Virtual Memory and I/O 16
2.4 Prototype Implementation 16
2.5 Preliminary Performance Evaluation 17
2.5.1 Sources of Overhead 18
2.5.2 Nanobenchmarks 18
2.5.3 Microbenchmarks 20
2.5.4 Macrobenchmarks 21
2.6 Related Work 21

Chapter 3 Secure Virtual Architecture **24**
3.1 Introduction 24
3.2 Overview of the SVA Approach 25
3.3 The SVA Execution Strategy 27
3.3.1 Instruction Set Characteristics 27
3.3.2 The SVA Boot and Execution Strategy 27
3.4 Implementations 28
3.5 Summary 29

Chapter 4 Memory Safety for a Commodity Operating System Kernel 30
4.1 Introduction 30
4.2 Overview of SVA-M 31
4.3 Enforcing Safety for Kernel Code 32
4.3.1 Background: How SAFECode Enforces Safety for C Programs 32
4.3.2 SAFECode for a Kernel: Challenges 35
4.3.3 Integrating Safety Checking with Kernel Allocators 36
4.3.4 Kernel Allocator Changes 39
4.3.5 Run-time Checks 40
4.3.6 Multiple Entry Points 41
4.3.7 Manufactured Addresses 42
4.3.8 Analysis Improvements 43
4.3.9 Summary of Safety Guarantees 44
4.4 Minimizing the Trusted Computing Base 46
4.5 Porting Linux to SVA 47
4.5.1 Porting to SVA-OS 47
4.5.2 Memory Allocator Changes 48
4.5.3 Changes to Improve Analysis 49
4.6 Experimental Results 50
4.6.1 Performance Overheads 50
4.6.2 Exploit Detection 54
4.6.3 Analysis Results 54
4.7 Related Work 56
4.8 Summary 58

Chapter 5 Secure Low-Level Software/Hardware Interactions 59
5.1 Introduction 59
5.2 Breaking Memory Safety with Low-Level Kernel Operations 62
5.2.1 Corrupting Processor State 63
5.2.2 Corrupting Stack State 64
5.2.3 Corrupting Memory-Mapped I/O 64
5.2.4 Corrupting Code 65
5.2.5 General Memory Corruption 66
5.3 Design Principles 66
5.4 Background: Secure Virtual Architecture 69
5.5 Design 70
5.5.1 Context Switching 70
5.5.2 Thread Management 71
5.5.3 Memory Mapped I/O 72
5.5.4 Safe DMA 73
5.5.5 Virtual Memory 73
5.5.6 Self-modifying Code 75
5.5.7 Interrupted State 76
5.5.8 Miscellaneous 77
5.6 Modifications to the Linux Kernel 77
5.6.1 Changes to Baseline SVA-M 77
5.6.2 Context Switching/Thread Creation 78
5.6.3 I/O 78
5.6.4 Virtual Memory 78
5.7 Evaluation and Analysis 79
5.7.1 Exploit Detection 79
5.7.2 Performance 81

5.8 Related Work . . . 85
5.9 Summary . . . 88

Chapter 6 Control-Flow Integrity for Operating System Kernels . . . 89
6.1 Introduction . . . 89
6.2 Attack Model . . . 91
6.3 KCoFI Infrastructure . . . 92
6.4 Design . . . 93
6.4.1 Control-flow Integrity Policy and Approach . . . 93
6.4.2 Protecting KCoFI Memory with Software Fault Isolation . . . 94
6.4.3 MMU Restrictions . . . 95
6.4.4 DMA and I/O Restrictions . . . 98
6.4.5 Thread State . . . 98
6.4.6 Protecting Interrupted Program State . . . 99
6.4.7 Thread Creation . . . 101
6.4.8 Context Switching . . . 102
6.4.9 Code Translation . . . 102
6.4.10 Installing Interrupt and System Call Handlers . . . 103
6.5 Formal Model and Proofs . . . 103
6.5.1 KCoFI Virtual Machine Model . . . 104
6.5.2 Instruction Set and Semantics . . . 107
6.5.3 Foundational Control-Flow Integrity Theorems . . . 108
6.5.4 Complete Control-Flow Integrity Theorems . . . 110
6.6 Implementation . . . 112
6.6.1 Instrumentation . . . 112
6.6.2 KCoFI Instruction Implementation . . . 113
6.6.3 Interrupt Context Implementation . . . 113
6.6.4 Unimplemented Features . . . 114
6.7 Security Evaluation . . . 114
6.7.1 Average Indirect Target Reduction . . . 114
6.7.2 ROP Gadgets . . . 115
6.8 Performance Evaluation . . . 116
6.8.1 Web Server Performance . . . 116
6.8.2 Secure Shell Server Performance . . . 117
6.8.3 Microbenchmarks . . . 117
6.8.4 Postmark Performance . . . 119
6.9 Related Work . . . 119
6.10 Summary . . . 121

Chapter 7 Protecting Applications from Compromised Operating Systems . . . 123
7.1 Introduction . . . 123
7.2 System Software Attacks . . . 126
7.2.1 Threat Model . . . 126
7.2.2 Attack Vectors . . . 127
7.3 Secure Computation Programming Model . . . 129
7.3.1 Virtual Ghost Memory Organization . . . 129
7.3.2 Ghost Memory Allocation and Deallocation . . . 130
7.3.3 I/O, Encryption, and Key Management . . . 131
7.3.4 Security Guarantees . . . 132
7.4 Enforcing Secure Computation . . . 133
7.4.1 Overview of Virtual Ghost . . . 133
7.4.2 Preventing Data Accesses in Memory . . . 133

7.4.3 Preventing Data Accesses During I/O . . . 135
7.4.4 Preventing Code Modification Attacks . . . 136
7.4.5 Protecting Interrupted Program State . . . 137
7.4.6 Mitigating System Service Attacks . . . 139
7.5 Implementation . . . 139
7.6 Securing OpenSSH . . . 142
7.7 Security Experiments . . . 143
7.8 Performance Experiments . . . 144
7.8.1 Microbenchmarks . . . 144
7.8.2 Web Server Performance . . . 145
7.8.3 OpenSSH Performance . . . 146
7.8.4 Client Performance With Ghosting . . . 147
7.8.5 Postmark Performance . . . 148
7.9 Related Work . . . 149
7.10 Summary . . . 151

Chapter 8 Conclusions and Future Work . . . 152

Appendix A Secure Virtual Architecture Instruction Set . . . 154
A.1 I/O . . . 154
A.1.1 sva_io_read . . . 154
A.1.2 sva_io_write() . . . 154
A.2 Interrupts . . . 155
A.2.1 sva_load_lif . . . 155
A.2.2 sva_save_lif . . . 156
A.3 Event Handlers . . . 156
A.3.1 sva_register_syscall . . . 156
A.3.2 sva_register_interrupt . . . 157
A.3.3 sva_register_general_exception . . . 158
A.3.4 sva_register_memory_exception . . . 159
A.4 Context Switching . . . 159
A.4.1 sva_swap_integer . . . 159
A.4.2 sva_load_fp . . . 160
A.4.3 sva_save_fp . . . 161
A.5 Interrupted Program State . . . 162
A.5.1 sva_was_privileged . . . 162
A.5.2 sva_icontext_lif . . . 162
A.5.3 sva_icontext_get_stackp . . . 163
A.5.4 sva_icontext_load_retval . . . 163
A.5.5 sva_icontext_save_retval . . . 163
A.5.6 sva_icontext_commit . . . 164
A.5.7 sva_icontext_push . . . 165
A.5.8 sva_icontext_save . . . 165
A.5.9 sva_icontext_load . . . 166
A.5.10 sva_ialloca . . . 166
A.5.11 sva_iunwind . . . 167
A.5.12 sva_init_icontext . . . 168
A.5.13 sva_reinit_icontext . . . 169
A.6 Exceptions . . . 169
A.6.1 sva_invoke_memcpy . . . 169
A.7 Bitcode Translation . . . 170
A.7.1 sva_translate . . . 170

A.8 Memory Management . . . 171
A.8.1 sva_mm_load_pagetable . . . 171
A.8.2 sva_mm_save_pagetable . . . 171
A.8.3 sva_mm_flush_tlbs . . . 172
A.8.4 sva_mm_flush_tlb . . . 172
A.8.5 sva_mm_flush_wcache . . . 173
A.8.6 sva_declare_ptp . . . 173
A.8.7 sva_release_ptp . . . 174
A.8.8 sva_update_l1_mapping . . . 175
A.8.9 sva_update_l2_mapping . . . 176
A.9 Virtual Ghost Application Instructions . . . 176
A.9.1 sva_alloc_ghostmem . . . 176
A.9.2 sva_free_ghostmem . . . 177
A.9.3 sva_validate_target . . . 178
A.9.4 sva_get_key . . . 178

Appendix B Control-Flow Integrity Proofs . . . 179
B.1 TLB.v . . . 179
B.2 Instructions.v . . . 180
B.3 Memory.v . . . 181
B.4 MMU.v . . . 184
B.5 Stack.v . . . 191
B.6 IC.v . . . 192
B.7 Thread.v . . . 193
B.8 Config.v . . . 196
B.9 Semantics.v . . . 204
B.10 ICText.v . . . 224
B.11 ICProofs.v . . . 238
B.12 InvProofs.v . . . 282
B.13 ThreadProofs.v . . . 284
B.14 ThreadTextProofs.v . . . 305
B.15 SVAOS.v . . . 333
B.16 Multi.v . . . 341
B.17 VG.v . . . 347

References . . . 354

List of Tables

2.1 Functions for Saving and Restoring Native Processor State . . . 12
2.2 Functions for Registering Interrupt, Trap, and System Call Handlers . . . 13
2.3 Functions for Manipulating the Interrupt Context . . . 14
2.4 System Call, Invoke, MMU, and I/O Functions . . . 15
2.5 Nanobenchmarks. Times in μs. . . . 19
2.6 High Level Kernel Operations. Times are in μs. . . . 20
2.7 Read Bandwidth for 4 MB file (bw_file_rd and bw_pipe). . . . 20
2.8 Web Server Bandwidth Measured in Megabits/second. . . . 21

4.1 Instructions for Registering Memory Allocations . . . 38
4.2 Number of lines modified in the Linux kernel . . . 48
4.3 Application latency increase as a percentage of Linux native performance. . . . 52
4.4 `thttpd` Bandwidth reduction . . . 52
4.5 Latency increase for raw kernel operations as a percentage of Linux native performance . . . 53
4.6 Bandwidth reduction for raw kernel operations as a percentage of Linux native performance . 54
4.7 Static metrics of the effectiveness of the safety-checking compiler . . . 55

5.1 SVA Instructions for Context Switching and Thread Creation. . . . 71
5.2 MMU Interface for a Hardware TLB Processor. . . . 74
5.3 Latency of Applications. Standard Deviation Shown in Parentheses. . . . 81
5.4 Latency of Applications During Priming Run. . . . 82
5.5 Latency of Kernel Operations. Standard Deviation Shown in Parentheses. . . . 85

6.1 KCoFI MMU Instructions . . . 94
6.2 KCoFI Interrupt Context Instructions . . . 97
6.3 KCoFI Context Switching Instructions . . . 100
6.4 KCoFI Native Code Translation Instructions . . . 100
6.5 Summary of Formal Model Support Functions . . . 106
6.6 LMBench Results. Time in Microseconds. . . . 118
6.7 LMBench: Files Creations Per Second . . . 119
6.8 Postmark Results . . . 119

7.1 Ghost Memory Management Instructions . . . 129
7.2 LMBench Results. Time in Microseconds. . . . 145
7.3 LMBench: Files Deleted Per Second. . . . 145
7.4 LMBench: Files Created Per Second. . . . 145
7.5 Postmark Results . . . 148

List of Figures

2.1 System Organization with SVA 8

3.1 System Organization with SVA 25

4.1 SVA-M System Organization 32
4.2 Points-to Graph Example 37

5.1 Web Server Bandwidth (Linux/i386 = 1.0) 82
5.2 SSH Server Bandwidth (Linux/i386 = 1.0) 83
5.3 File System Bandwidth 84

6.1 SVA/KCoFI Architecture 92
6.2 KCoFI Address Space Organization 95
6.3 KCoFI Thread Structure 99
6.4 Instructions in KCoFI Model 104
6.5 KCoFI Instruction Semantics 109
6.6 ApacheBench Average Bandwidth with Standard Deviation Bars 117
6.7 SSHD Average Transfer Rate with Standard Deviation Bars 118

7.1 System Organization with Virtual Ghost 134
7.2 Average Bandwidth of thttpd 146
7.3 SSH Server Average Transfer Rate 147
7.4 Ghosting SSH Client Average Transfer Rate 148

Chapter 1

Introduction

1.1 Motivation

Today, computer users trust commodity operating system kernels, such as Windows, Mac OS X, Linux, and FreeBSD, with their sensitive data. Using applications that run on these operating system kernels, users buy items over the Internet from services such as Amazon [5] and iTunes [6] and pay their taxes online using services such as TurboTax [9]. Some voting machines run commodity operating systems (e.g., Windows CE [65]).

While computer users now rely on commodity operating systems, such reliance comes at great peril. Commodity operating system designs permit the operating system kernel to have access to *all* system resources, including physical memory used by the kernel and applications, I/O devices, and processor configuration [119, 130, 30, 99]. Furthermore, commodity operating system kernels are susceptible to a number of vulnerabilities. Some of these vulnerabilities are due to the fact that operating system kernels are written in C and C++ and therefore suffer from memory safety vulnerabilities like buffer overflows [17] and dangling pointer attacks [14] which can be used to subvert control-flow [132, 118] and data-flow [37]. Other vulnerabilities, such as kernel-level malware [84], are due to kernel-specific functionality (for example, the fact that kernels can modify their behavior at run-time using dynamically loaded modules). These vulnerabilities are not just theoretical: they have been discovered and documented as real problems in commodity operating system kernels [94, 21, 102, 137, 72, 135, 136, 137, 146, 7, 42, 43, 133].

Because they are so highly privileged, commodity operating system kernel vulnerabilities are a serious threat. If a commodity operating system kernel is exploited and controlled by an attacker, then *all* security policy enforcement on the system, including enforcement provided by applications running on the kernel, can be bypassed.

This work explores two general approaches to solving the problem of vulnerable commodity operating system kernels. The first approach is to automatically harden the operating system kernel from attack, and

this work explores two different approaches to hardening the operating system kernel from memory safety attacks. A limitation of this first approach is that it can only defend against a single class of attacks (e.g., memory safety errors). Therefore, this work explores a second approach: limit how an operating system interacts with applications and the hardware so that an application can continue to operate securely even when the operating system kernel is compromised by an arbitrary exploit. This second approach is more holistic; it can protect applications regardless of how the operating system kernel is compromised.

1.2 Secure Virtual Architecture

Any solution for securing systems that use commodity operating system kernels imposes several requirements. First, it must be able to enforce security policies on both operating system and application code. Second, it must be able to analyze the interactions between an application and an operating system to ensure that one does not corrupt the other. Finally, in order to gain widespread acceptance, a solution must require minimal modification of the software stack.

For the security policies explored in this work, a fourth requirement emerges: the solution must be able to use compiler analysis and instrumentation techniques on operating system kernel code. Applying such techniques is challenging: commodity operating systems have assembly code that is difficult to analyze and instrument reliably, and it can be difficult for compiler techniques to analyze the interactions between the operating system kernel and applications.

This work proposes and evaluates one such solution that meets these requirements. Based on virtual instruction set computing [13], this solution, called the *Secure Virtual Architecture* (SVA), interposes a compiler-based virtual machine between the software stack and the hardware. Software is encoded in a *virtual instruction set* that enables sophisticated compiler analysis techniques to be applied to the software; the virtual machine then translates the virtual instructions to native instructions for execution on the processor [13]. SVA provides a minimal amount of abstraction for the hardware; this makes the porting effort to SVA minimal and permits the operating system to maintain control over resource allocation decisions. With the ability to use compiler technology, SVA can analyze and transform software code before execution, employing previously proven security enforcement techniques to provide memory safety [57] and control-flow integrity [10]. Because SVA can control how the operating system (OS) kernel interacts with applications and the hardware, it can also be used to enforce other security policies that protect applications from the OS kernel without the use of hardware privilege levels higher than the one used for the OS kernel.

1.3 Enforcing Security Policies with SVA

This work describes the design of SVA and explores three different security policies implemented with SVA. SVA provides a solid foundation for enforcing security policies. With its compiler support, SVA can enforce policies that require static analysis or program instrumentation capabilities (even for OS kernel code). By being placed underneath the software stack, SVA can also utilize privileged hardware features, and it can use them in conjunction with compiler-based security techniques. Finally, SVA makes the interactions between the OS kernel and both the hardware and applications explicit, thereby supporting policies that need to accurately analyze and control such interactions.

All three policies explored in this work employ the compiler capabilities that SVA provides for OS kernel code as well as SVA's ability to control how the OS kernel configures the hardware. In addition, all three security policies utilize SVA's ability to analyze the interactions between the OS kernel and applications.

The first policy is *strong memory safety* and is based on previously developed compiler transformations [57, 56] for user-space applications. These memory safety techniques provide control-flow integrity, ensure that loads and stores only access memory objects belonging to the correct points-to set, prevent pointers from "jumping" from one valid memory object into another, and provide type-safety for a subset of memory objects. This policy can stop numerous memory safety attacks, including control-flow hijack attacks [132, 118] and non-control data attacks [37].

The second policy that this work explores is control-flow integrity [10]. Control-flow integrity is simpler to implement than memory safety and incurs less overhead. Like memory safety, control-flow integrity can prevent control-flow hijack attacks. The price to pay for this performance is security; control-flow integrity cannot stop non-control data attacks as memory safety can.

The third and final policy that this work explores is application protection. As stated earlier, a shortcoming of memory safety and control-flow integrity is that they only address a small (but important) class of attacks; they do not, for example, prevent attacks that can be mounted by kernel rootkits [84]. This work therefore develops a new compiler-enforced security policy that enables an application to preserve the confidentiality and integrity of its data *even if the OS kernel has been compromised*. This policy is more holistic; any application that is written to distrust the OS kernel is protected from several different attack classes.

1.4 Contributions

This work makes the following novel contributions:

- It extends the *virtual instruction set* originally defined by Adve et. al. [13] with instructions needed by commodity operating systems, hypervisors, and utility library code. These instructions replace hand-written assembly code that performs operations such as context switching, MMU management, and thread management. The virtual instruction set is sufficiently low-level as to keep policy decisions within commodity software while allowing security policies to be efficiently enforced using compiler-based strategies. Porting a commodity operating system to this virtual instruction set requires minimal effort. To date, Linux 2.4.22 and FreeBSD 9.0 have been ported to the virtual instruction set.

- It describes how, using SVA, a set of previously developed compiler transformations [57, 56] can be adapted to enforce strong memory safety guarantees on operating system kernel code. It adapts these techniques to handle existing kernel memory allocators, multiple kernel entry points, and programming techniques typically used only in kernel code.

- It describes an analysis and redesign of the virtual instruction set to handle memory safety violations that can occur through misuse of low-level software/hardware interactions such as context switching, MMU configuration, and thread creation.

- It describes an implementation of SVA called *KCoFI* (pronounced "coffee") which enforces a *complete* control-flow integrity policy on OS kernel code. A subset of the SVA instruction set is modeled using a state transition system, and a formal proof shows that the restrictions imposed by KCoFI enforce control-flow integrity.

- It describes an SVA implementation called *Virtual Ghost* that provides a feature to applications called *Ghost Memory*. Ghost memory is memory which the OS kernel cannot read or write. Combined with other new features described herein, Virtual Ghost can prevent a compromised OS kernel from stealing or corrupting an application's data and can also prevent the operating system from subverting an application's control-flow.

1.5 Organization

The remainder of this work is organized as follows:

Chapter 2 is derived from one of our papers [51] and describes the basic system organization of SVA and how we created a virtual instruction set that supports commodity operating system code. Chapter 3, partially based on previous work [50], gives an overview of how we use the virtual instruction set from Chapter 2 to enforce security policies on operating system code. Chapter 4, based on our previous work [50], describes how we use the SVA virtual machine to enforce memory safety on operating system code, and Chapter 5, based on one of our SVA papers [49], addresses the special memory safety issues that arise in low-level interactions with the hardware. Chapter 6 extends our previous work [47] and describes how the SVA features can be used to implement complete control-flow integrity for a commodity operating system kernel. Chapter 7 extends the system in Chapter 6 and describes modifications to SVA that protect application computation from errant (and potentially malicious) operating system code; it is based on our recent work [48]. Chapter 8 concludes by summarizing the current work on SVA and describes how SVA can enable future research.

This work also includes two appendices. Appendix A describes the SVA-OS instruction set. Appendix B contains the Coq code that, when mechanically checked by the Coq proof assistant, proves the control-flow integrity theorems from Chapter 6.

Chapter 2

Virtual Architecture Support for Operating System Kernels

2.1 Introduction

Modern operating system (OS) kernels are compiled into machine code and use a set of low-level hardware instructions that allows the kernel to configure the OS response to hardware events and to manipulate program state. Because of these choices, substantial parts of the kernel are difficult to analyze, type-check, or verify. For example, even basic but crucial properties like memory safety become difficult to enforce. Program analysis techniques for more sophisticated security properties (e.g., enforcing isolation between kernel extensions and the core kernel [141] or analyzing reference checks on sensitive kernel operations [67]) are also handicapped. First, they can only be applied to a limited extent because of the presence of inline assembly code. Second, they must be applied offline because it is difficult to analyze machine code, which means they cannot be used at key moments like loading a kernel extension or installing a privileged program. In practice, such compiler techniques are simply not applied for widely-used legacy systems like Linux or Windows.

An alternative approach that could ameliorate these drawbacks and enable novel security mechanisms is to compile kernel code to a rich, virtual instruction set and execute it using a *compiler-based* virtual machine. Such a virtual machine would incorporate a *translator* from virtual to native code and a run-time system that monitors and controls the execution of the kernel. To avoid the performance penalties of dynamic compilation, the translation does not need to happen online: it can be performed offline and cached on persistent storage.

Previous work presents a virtual instruction set called LLVA (Low Level Virtual Architecture) [13] that is sufficiently low-level to support arbitrary programming languages (including C) but rich enough to enable sophisticated analyses and optimizations. LLVA provides computational, memory access, and control flow operations but lacks operations an OS kernel needs to configure hardware behavior and manipulate program state.

This chapter creates a variant of LLVA called *Secure Virtual Architecture*, or *SVA* that will be used in later chapters to enforce strong security policies on software. This variant extends the original LLVA instruction set with instructions collectively called SVA-OS. SVA-OS provides an interface between the OS kernel and a general purpose processor architecture. An *Execution Engine* translates SVA code to machine code and includes a library that implements the SVA-OS operations. Together, SVA and the Execution Engine define a virtual machine capable of hosting a complete, modern kernel, and we will use them in later chapters to enforce powerful security guarantees on operating system code. We also observe that kernel portability is not a primary goal of this work even though it may be achieved as a side effect of the virtual instruction set design.

The primary contributions of this chapter are:

1. The design of SVA-OS, including novel primitive mechanisms for supporting a kernel that are higher-level than traditional architectures.

2. A prototype implementation of SVA-OS and a port of the Linux 2.4.22 kernel to SVA-OS.

3. A preliminary evaluation of the prototype that shows the performance overhead of virtualization in terms of four root causes.

Our evaluation revealed where our design choices added virtualization overhead to the Linux kernel: context switching, data copies between user and kernel memory, read page faults, and signal handler dispatch. Our analysis explores how certain design decisions caused overhead in page faults and data copying and how better implementation could reduce context switching and data copying overheads. Our analysis also shows that many of these overheads do not severely affect overall performance for a small set of applications, and that, in most cases, these overheads can be reduced with relatively simple changes to the SVA-OS design or implementation.

Section 2.2 describes in more detail the system organization we assume in this work. Section 2.3 describes the design of the SVA-OS interface. Section 2.4 briefly describes our experience implementing SVA-OS and porting a Linux kernel to it. Section 2.5 evaluates the performance overheads incurred by the kernel on the SVA-OS prototype, and Section 2.6 compares SVA-OS with previous approaches to virtualizing the OS-hardware interface.

2.2 Background: VISC Architectures and LLVA

Our virtual instruction set extensions are based upon LLVA [13], illustrated in Figure 2.1. This instruction set is implemented using a virtual machine that includes a translator (a code generator), a profile-guided optimizer, and a run-time library used by translated code. This virtual machine is called the Low Level Execution Engine (LLEE).

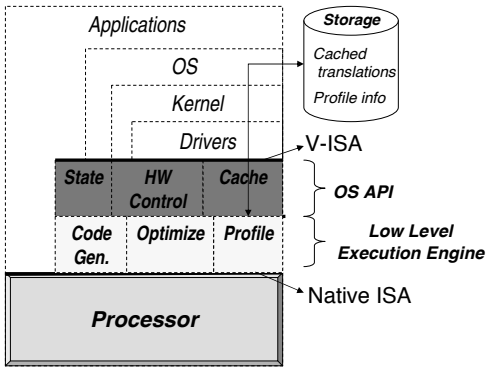


Figure 2.1: System Organization with SVA

The functionality provided by our new virtual instruction set can be divided into two parts:

- *Computational interface*, i.e., the core instruction set;
- *Operating System interface*, or SVA-OS;

The computational interface, defined in previous work [13], is the core instruction set used by all SVA software for computation, control flow, and memory usage. It is a RISC-like, load-store instruction set with ordinary arithmetic and logical instructions, comparison instructions that produce a boolean value, explicit conditional and unconditional branches, typed instructions for indexing into structures and arrays, load and store instructions, function call instructions, and both heap and stack memory allocation and deallocation instructions. Heap objects can be allocated using explicit `malloc` and `free` instructions that are (typically) lowered to call the corresponding standard library functions; however, some heap objects may be allocated via other functions (e.g., custom allocators) that may not be obvious to the compiler. The instruction set also supports fundamental mechanisms for implementing exceptions (including C++ exceptions and C's `setjmp/longjmp`) and multithreaded code. For SVA, we extended the LLVA instruction

set with atomic memory access instructions (atomic load-increment-store and compare-and-swap) and a memory write barrier instruction.

Compared with traditional machine instruction sets, LLVA (and SVA) includes four novel features: a language-neutral type system suitable for program analysis, an infinite register set in static single assignment (SSA) form [52], an explicit control flow graph per function, and explicit distinctions between *code*, *stack*, *globals*, and *other* memory. These features allow extensive analysis and optimizations to be performed directly on LLVA/SVA code, either offline or online [13].

The OS interface, SVA-OS, is the focus of this chapter. This interface provides a set of operations that is primarily used by the OS to control and manipulate architectural state.

2.3 Design of the OS Interface

Operating systems require two types of support at the OS-hardware interface. First, the OS needs to access specialized and privileged hardware components. Such operations include registering interrupt handlers, configuring the MMU, and performing I/O. Second, the OS needs to manipulate the state of itself and other programs, e.g. context switching, signal delivery, and process creation.

2.3.1 Design Goals for the OS Interface

To achieve our primary long term goal of designing an architecture that improves the security and reliability of system software, we designed SVA-OS with the following goals:

- The SVA-OS must be designed as a set of abstract (but primitive) operations, independent of a particular hardware ISA.
- The SVA-OS must be OS neutral (like current native instruction sets) and should not constrain OS design choices.
- The SVA-OS must be light weight and induce little performance overhead.

The first goal is beneficial to enhance kernel portability across some range of (comparable) architectures. Although it may be similar to a well-designed portability layer within the kernel, moving this into the virtual instruction set can expose greater semantic information about kernel operations to the underlying translator and even the hardware. For example, it can allow the translator to perform sophisticated memory safety checks for the kernel by monitoring page mappings, system calls, and state-saving and restoring

operations. Keeping SVA-OS independent of the hardware ISA also allows SVA to utilize new processor features transparently; the software at the virtual instruction set level does not need to be modified to take advantage of new processor features. The second goal allows our work to be applied to a variety of operating systems used in a wide range of application domains. The third goal is important because our aim is to incorporate the virtual machine model below widely-used, legacy operating systems where any significant performance overhead would be considered unacceptable.

2.3.2 Structure of the Interface

We define SVA-OS, semantically, as a set of functions, i.e., an API, using the syntax and type system of the core SVA instruction set. We call these functions *intrinsic functions*, or intrinsics, because their semantics are predefined, like intrinsics in high-level languages. When compiling a call to such a function, the translator either generates inline native code for the call (effectively treating the function like a virtual instruction) or generates a call to a run-time library within the Execution Engine containing native code for the function.

Using function semantics for the interface, instead of defining it via explicit SVA instructions, provides two primary benefits. First, uses of these functions appear to a compiler as ordinary (unanalyzable) function calls and therefore do not need to be recognized by any compiler pass except those responsible for translation to native code. Second, using functions makes the control flow of the operations more explicit, since most operations behave like a called function: some code is executed and then control is returned to the caller.

2.3.3 Virtual and Native System State

The state of an executing program in SVA-OS can be defined at two levels: virtual state and native state. Virtual state is the system state as seen by external software, including the OS. The virtual state of an SVA program includes:

- The set of all virtual registers for all currently active function activations.
- The implicit program counter indicating which virtual instruction to execute next.
- The contents of memory used by the current program.
- The current stack pointer indicating the bottom of the currently active stack.
- The current privilege mode of the processor (either privileged or unprivileged).
- A set of MMU control registers.
- A local interrupt enable/disable flag.

Native state is the state of the underlying physical processor and includes any processor state used by a translated program, such as general purpose, floating point, and control flow registers. It may also include the state of co-processors, such as MMUs or FPUs.

A key design choice we made in SVA-OS is to expose the *existence* of native state while keeping the *contents* hidden. In particular, we provide intrinsics that can save and restore native state without being able to inspect or manipulate the details of native state directly. This design choice is motivated and explained in Section 2.3.4.

2.3.4 Manipulating Virtual and Native State

There are two broad classes of operations that a program may use to manipulate the state of *another, separately compiled, program*: (i) saving and restoring the entire state of a program, e.g., an OS context switching routine, and (ii) directly inspecting or modifying the details of program state, e.g., by a debugger.

Some virtual machines, e.g., the Java Virtual Machine (JVM) [93], encapsulate the saving and restoring of state entirely inside the VM. This yields a simple external interface but moves policy decisions into the VM. This model is inadequate for supporting arbitrary operating systems without significant design changes to allow external control over such policies. Other virtual machines, such as the Transmeta [55] and DAISY [61] architectures based on binary translation, allow external software to save or restore virtual program state but hide native state entirely. This requires that they maintain a mapping at all times between virtual and native state. Maintaining such a mapping can be expensive without significant restrictions on code generation because the mapping can change frequently, e.g., every time a register value is moved to or from the stack due to register spills or function calls.

We propose a novel solution based on two observations. First, operations of class (i) above are frequent and performance-sensitive while those of (ii) are typically not. Second, for class (i), an OS rarely needs to inspect or modify individual virtual registers in order to save and restore program state. We provide a limited method of allowing the OS to directly save and restore the native state of the processor to a memory buffer. The native state is visible to the OS only as an opaque array of bytes.

For operations of class (ii), the translator can reconstruct the virtual to native mapping lazily when required. This moves the mapping process out of the critical path.

We provide the functions in Table 2.1 to save and restore native state. When designing these functions, we observed that the Linux kernel takes advantage of an important application property: floating point registers do not always need to be saved but integer registers usually do [30]. Therefore, we divide native processor

state into two sets. The first set is the Integer State. It contains all native processor state used to contain non-floating point virtual registers and the virtual software's control flow. On most processors, this will include all general purpose registers in use by the program, the program counter, the stack pointer, and any control or status registers. The second set is the Floating Point (FP) State. This set contains all native state used to implement floating point operations. Usually, this is the CPU's floating point registers. Additional sets can be added for state such as virtual vector registers (usually represented by vector co-processors in native state). Native state not used by translated code, e.g. special support chips, can be accessed using I/O instructions (just as they are on native processors) using the I/O instructions in Table 2.4.

Name	**Description**
sva.save.integer(void * buffer)	Save the Integer State of the native processor in to the memory pointed to by *buffer*.
sva.load.integer(void * buffer)	Load the integer state stored in *buffer* back on to the processor. Execution resumes at the instruction immediately following the sva.save.integer() instruction that saved the state.
sva.save.fp(void * buffer, int always)	Save the FP State of the native processor or FPU to the memory pointed to by *buffer*. If *always* is 0, state is only saved if it has changed since the last sva.load.fp(). Otherwise, save state unconditionally.
sva.load.fp(void * buffer)	Load the FP State of the native processor (or FPU) from a memory buffer previously used by sva.save.fp().

Table 2.1: Functions for Saving and Restoring Native Processor State

2.3.5 Interrupts and Traps

SVA-OS defines a set of interrupts and traps identified by number. These events are serviced by handler functions provided by the OS. SVA-OS provides intrinsics that the OS uses to register handler functions for each interrupt or trap; these are shown in Table 2.2. On an interrupt or trap, a handler function is passed the interrupt or trap number and a pointer to an Interrupt Context, defined below. Page fault handlers are passed an additional address parameter.

When an interrupt or trap occurs, the processor transfers control to the Execution Engine, which saves the native state of the interrupted program (on the kernel stack) before invoking the interrupt or trap handler. Saving the entire Integer State would be sufficient but not always necessary. Many processors provide mechanisms, e.g., shadow registers, to reduce the amount of state saved on an interrupt.

To take advantage of such hardware, when available, we define the *Interrupt Context*: a buffer of memory reserved on the kernel stack capable of holding the complete Integer State when an interrupt or trap occurs.

Name	Description
sva.register.interrupt (uint number, void (*f)(uint, void *))	Register a function as an interrupt handler for the given interrupt number. The interrupt handler is given the interrupt number and a pointer to the Interrupt Context.
sva.register.general.trap (uint number, void (*f)(void *))	Register a function as a trap handler for the given trap number. The trap handler is passed a pointer to the Interrupt Context.
sva.register.memory.trap (uint number, void (*f)(void *, void *))	Register a function as a memory trap handler. The trap handler is passed a pointer to the Interrupt Context and a pointer to the memory location that caused the trap.
sva.register.syscall (uint number, void (*f)(...))	Register a function as a system call handler. The system call handler is passed a pointer to the Interrupt Context and the arguments passed into the system call from the program invoking the system call.

Table 2.2: Functions for Registering Interrupt, Trap, and System Call Handlers

Only the subset of Integer State that will be overwritten by the trap handling code need be saved in the reserved memory by the Execution Engine. Any other part of Integer State masked by facilities such as shadow registers is left on the processor.

In cases where the complete Interrupt Context must be committed to memory, e.g., context switching or signal handler dispatch, we provide intrinsics that can commit all the Integer State inside of an Interrupt Context to memory. This allows us to lazily save interrupted program state when needed. Interrupt and trap handlers can also use the Interrupt Context to manipulate the state of an interrupted program. For example, an OS trap handler can push a function call frame on to an interrupted program's stack using sva.ipush.function(), forcing it to execute a signal handler when the trap handler finishes. Table 2.3 summarizes the various functions that manipulate the Interrupt Context.

2.3.6 System Calls

SVA-OS provides a finite number of system calls identified by unique numbers. Similar to interrupt and trap handlers, the OS registers a system call handler function for each system call number with an SVA-OS intrinsic function.

Name	Description
sva.icontext.save (void * icp, void * isp)	Save the Interrupt Context *icp* into the memory pointed to by *isp* as Integer State.
sva.icontext.load (void * icp, void * isp)	Load the Integer State *isp* into the Interrupt Context pointed to by *icp*.
sva.icontext.commit (void* icp)	Commit the entire Interrupt Context *icp* to memory.
sva.icontext.get.stackp (void * icp)	Return the stack pointer saved in the Interrupt Context.
sva.icontext.set.stackp (void * icp)	Set the stack pointer saved in the Interrupt Context.
sva.ipush.function (void * icp, int (*f)(...), ...)	Modify the state in the Interrupt Context *icp* so that function f has been called with the given arguments. Used in signal handler dispatch.
sva.icontext.init (void * icp, void * stackp)	Create a new Interrupt Context on the stack pointed to by *stackp*. It is initialized to the same values as the Interrupt Context *icp*.
sva.was.privileged (void * icp)	Return 1 if the Interrupt Context *icp* was running in privileged mode. Return 0 otherwise.

Table 2.3: Functions for Manipulating the Interrupt Context

Software initiates a system call with the sva.syscall intrinsic (Table 2.4). Semantically, this appears much like a function call. They only differ in that system calls switch to the privileged processing mode and call a function within the OS.

Unlike current designs, an SVA processor knows the semantic difference between a system call and an instruction trap and can determine the system call arguments. This extra information allows the same software to work on different processors with different system call dispatch mechanisms, enables the hardware to accelerate system call dispatch by selecting the best method for the program, and provides the Execution Engine and external compiler tools the ability to easily identify and modify system calls within software.

2.3.7 Recovery from Hardware Faults

Some operating systems use the MMU to efficiently catch errors at runtime, e.g., detecting bad pointers passed to the write system call [30]. The OS typically allows the page fault handler to adjust the program counter of the interrupted program so that it executes exception-handling assembly code immediately after the page fault handler exits. In an SVA-supported kernel, the OS cannot directly change the program counter (virtual or native), or write assembly-level fault handling code. Another mechanism for fault recovery is needed.

We observe that recovering from kernel hardware faults is similar to exception handling in high level languages. The SVA instruction set [86] provides two instructions (invoke and unwind) to support such exceptions. We adapted these instructions to support kernel fault recovery in SVA-OS.

Name	Description
int sva.syscall (int sysnum, ...)	Request OS service by calling the system call handler associated with number *sysnum*.
int sva.invoke (int * ret, int (*f)(...), ...)	Call function *f* with the specified arguments. If control flow is unwound before *f* returns, then return 1; otherwise, return 0. Place the return value of *f* into the memory pointed to by *ret*.
int sva.invokememcpy (void * to, void * from, int count)	Copy *count* bytes from *from* to *to*. Return the number of bytes successfully copied before a fault occurs.
sva.iunwind (void * icp)	Modify the state in the Interrupt Context pointed to by *icp* so that control flow is unwound to the innermost frame in which an invoke was executed.
void sva.load.pgtable (void * pgtable)	Load the page table pointed to by *pg*.
void * sva.save.pgtable (void)	Return the page table currently used by the MMU.
void sva.flush.tlbs (int global)	Flush the TLBs. If *global* is 1, remove global TLB entries.
void sva.flush.tlb (void * addr)	Flush any TLBs that are contain virtual address *addr*.
int sva.mm.protection (void * icp)	Return 1 if the memory fault was caused by a protection violation.
int sva.mm.access.type (void * icp)	Returns 1 if memory access was a read; 0 if it was a write.
int sva.mm.was.absent (void * icp)	Returns 1 if the memory access faulted due to translation with an unmapped page.
int sva.ioread (void * ioaddress)	Reads a value from the I/O address space.
void sva.iowrite (int value, void * ioaddress)	Writes a value into the I/O address space.

Table 2.4: System Call, Invoke, MMU, and I/O Functions

The invoke intrinsic (described in Table 2.4) is used within the kernel when calling a routine that may fault; the return value can be tested to branch to an exception handler block. Invokes may be nested, i.e., multiple invoke frames may exist on the stack at a time.

If the called function faults, the trap handler in the OS calls sva.iunwind. This function unwinds control flow back to the closest invoke stack frame (discarding all intervening stack frames) and causes the invoke intrinsic to return 1. The only difference between the original unwind instruction and sva.iunwind is that the latter unwinds the control flow in an Interrupt Context (to return control to the kernel context that was executing when the fault occurred) while the former unwinds control flow in the current context.

This method uses general, primitive operations and is OS neutral. However, its performance depends on efficient code generation of invoke by the translator [86]. Our current version of LLEE has implementation and design deficiencies as described in Section 2.5.2.

To address these, we added a combined invoke/memcpy intrinsic named sva.invokememcpy. This intrinsic uses efficient native instructions for data copying and always returns the number of bytes successfully copied (even if control flow was unwound by sva.iunwind).

2.3.8 Virtual Memory and I/O

MMUs vary significantly between different processor families, making a universal set of MMU intrinsics that allow the OS to take maximum advantage of the MMU features quite difficult to design. We have instead chosen the following strategy: when a processor family is first designed, an MMU model is selected, e.g. software TLB with explicit memory regions. The virtual interface to the MMU is then designed to take advantage of the MMU's features but hides the exact configuration details. This approach should allow the underlying MMU hardware to change without requiring changes in the OS.

One key distinction is that most MMUs follow one of two main design strategies for TLB miss handling: a hardware page table, e.g., x86 or PowerPC, or a software-managed TLB, e.g., Sparc, Alpha, and Itanium. An orthogonal design issue is that the MMU can optionally use memory regions like those found on the Itanium [46] and PowerPC [107].

Our current SVA-OS design assumes a hardware page table with no explicit memory regions. Intrinsics for changing the page table pointer and analyzing the cause of page faults are described in Table 2.4. We have yet to design intrinsics for abstracting the page table format; that is left as future work.

The I/O functions, used to communicate with I/O devices and support chips, are described in Table 2.4. The LLEE's code generator is responsible for inserting the necessary machine instructions (such as memory fence instructions) when translating the I/O virtual instructions to native code. The I/O address space may overlap with the memory address space (if devices are memory mapped), be a separate address space (if special instructions or addressing modes are needed to access I/O devices) or some combination thereof (for a machine that supports both memory mapped I/O and I/O port I/O). In our design, the I/O address space maps one to one with the memory address space except for a small region that is used for addresses that refer to x86 I/O ports.

2.4 Prototype Implementation

While designing SVA-OS, we implemented a prototype Execution Engine and ported the Linux 2.4.22 kernel to our virtual instruction set. This essentially worked as a port of Linux to a new instruction set. It took approximately one-person-year of effort to design and implement SVA-OS and to port Linux to it.

Our Execution Engine is implemented as a native code library written in C and x86 assembly. It can be linked to an OS kernel once the kernel has been compiled to native code. It provides all of the functionality described in Section 2.3 and does not depend on any OS services.

Compilation to the SVA instruction set is done using LLVM [86]. Since the LLVM compiler currently requires OS services to run, all code generation is performed ahead of time.

To port the Linux kernel, we removed all inline assembly code in i386 Linux and replaced it with C code or C code that used SVA-OS.

During the first phase of development, we continued to compile the kernel with GCC and linked the Execution Engine library into the kernel. This allowed us to port the kernel incrementally while retaining full kernel functionality. This kernel (called the SVA GCC kernel below) has been successfully booted to multi-user mode on a Dell OptiPlex, works well enough to benchmark performance, and is capable of running many applications, including the thttpd web server [115], GCC, and most of the standard UNIX utilities.

We have also successfully compiled the SVA-ported kernel with the LLVM compiler [86], demonstrating that the kernel can be completely expressed in the SVA instruction set. This kernel also boots into multi-user mode on real hardware.

2.5 Preliminary Performance Evaluation

We performed a preliminary performance evaluation on our prototype to identify key sources of overhead present in our current design. To do this, we benchmarked the SVA GCC and native i386 Linux kernels. We ran all of our tests on a Dell OptiPlex with a 550 MHz Pentium 3 processor and 384 MB of RAM. Since both kernels are compiled by the same compiler (GCC) and execute on identical hardware, the difference between them is solely due to the use of SVA-OS as the target architecture in the former. The use of this interface produces multiple sources of overhead (described below) because a kernel on SVA-OS uses *no native assembly code*, whereas the original kernel uses assembly code in a number of different ways. It is important to note that the use of the interface alone (with no additional hardware or translation layer optimizations) does not improve performance in any way, except for a few incidental cases mentioned below.

We do not compare the SVA kernel compiled with LLVM to the above two kernels. Doing so compares the quality of code generated by the LLVM and GCC compilers which, while useful long-term, does not help us identify sources of overhead in the SVA-OS design.

We used three different types of benchmarks to study the performance impact of SVA-OS. Nano benchmarks test primitive kernel functions, e.g., system call and trap latency, that typically use only a few SVA-OS intrinsics. These overheads can be classified according to one of the four causes below. Micro benchmarks measure the performance of high-level kernel operations directly used by applications, such as specific system calls. Finally, macro benchmarks are entire applications and measure the performance impact of the

abstractions on overall system performance. Unless stated otherwise, all benchmarks use the HBench-OS framework [31] and present the average measurement of 100 iterations.

2.5.1 Sources of Overhead

There are four distinct causes of overhead when a kernel uses the SVA-OS virtual architecture on a particular processor, compared with an identical kernel directly executing on the same hardware:

1. The native kernel used assembly code to increase performance and the same operation on SVA must be written using C code, e.g. IP Checksum code on x86 can exploit the processor status flags to check for overflow, saving a compare instruction.

2. The native kernel used assembly code for an operation (for performance or hardware access) and SVA-OS provides an equivalent function, but the function is not implemented efficiently.

3. The native kernel used assembly code to exploit a hardware feature, and this feature is less effectively used in the SVA-OS design, i.e., a mismatch between SVA-OS design and hardware.

4. The native kernel exploited OS information for optimizing an operation, and this optimization is less effective with SVA-OS, i.e., a mismatch between SVA-OS design and kernel design.

These sources of overhead are important to distinguish because the solution for each is different. For example, the first two sources above have relatively simple solutions: the first by adding a new intrinsic function to SVA; the second simply by tuning the implementation. Mismatches between SVA-OS design and either hardware features or OS algorithms, however, are more difficult to address and require either a change in SVA-OS or in the design of the OS itself.

2.5.2 Nanobenchmarks

We used the benchmarking software from [104] to measure the overhead for a subset of primitive kernel operations (Table 2.5). These tests are based on HBench-OS tests [31] and use specialized system calls in the kernel to invoke the desired nano-benchmark feature. Many other primitive operations, particularly synchronization and bit manipulation primitives, are presented in [104].

As seen in Table 2.5, only a few primitive operations incur significant overhead on SVA-OS. Note that these operations are extremely short (microseconds). Small inefficiencies produce large relative overhead, but their impact on higher-level kernel operations and applications is usually far smaller.

Operation	Native	SVA	% Overhead	SVA-OS Intrinsics
System Call Entry	.589	.632	7.30	sva.syscall
Trap Entry	.555	.450	-18.92	Internal to LLEE
Read Page Fault	1.105	1.565	41.63	sva.mm.protection, sva.mm.access.type, sva.mm.was.absent
Kernel-User 1KB memcpy	.690	.790	14.45	sva.invokememcpy
User-Kernel 1KB strcpy	.639	.777	21.64	invoke

Some numbers from [104].

Table 2.5: Nanobenchmarks. Times in μs.

We improve performance for trap entry. The i386 kernel supports traps from Virtual 8086 mode. The SVA kernel does not, yielding simpler trap entry code.

The slight increase in system call latency is because the native kernel saves seven fewer registers by knowing that they are not modified by system call handlers [104]. Some of the overhead is also due to some assembly code (which initiates signal dispatch and scheduling after every system call) being re-written in C. The overhead is partially due to a mismatch between SVA-OS design and Linux design and partially due to the inability to write hand-tuned assembly code. A read page fault has relative overhead of 42% (but the absolute difference is tiny) [104] and is a mismatch between SVA-OS and hardware. The native Linux page fault handler uses a single bitwise operation on a register to determine information about the page fault; the SVA kernel must use two SVA-OS intrinsics to determine the same information [104].

The overhead for copying data between user and kernel memory stems from several sources. Both the invoke instruction (used by strcpy) and sva.invokememcpy (used by memcpy) save a minimal amount of state so that sva.iunwind can unwind control flow. The invoke instruction adds function call overhead since it can only invoke code at function granularity. Finally, the invoked strcpy function is implemented in C. The i386 kernel suffers none of these overheads.

Monroe [104] found that, for a small number of processes, context switching latency doubled with the SVA kernel due to SVA-OS saving more registers than the original context switching code. However, once the system reaches 60 active processes, the overhead of selecting the next process to run is greater than the SVA-OS overhead due to a linear time scan of the run queue [104, 30]. For a large number of processes, SVA-OS only adds 1-2 microseconds of overhead.

2.5.3 Microbenchmarks

We used the HBench-OS benchmark suite [31] to measure the latency of various high level kernel operations. We also used a test developed by Monroe [104] that measures the latency of opening and closing a file repeatedly. Tables 2.6 and 2.7 present the results.

All the system calls listed in Tables 2.6 and 2.7 incur the system call entry overhead in Table 2.5. All of the I/O system calls incur the memcpy overhead, and any system call with pathname inputs incurs the strcpy overhead.

Operation	**Program**	**Native**	**SVA**	**% Overhead**
open / close	N/A	4.43	5.00	12.93
fork and exit	lat_proc null	268.25	279.19	4.08
fork and exec	lat_proc simple static	1026	1100	7.25
Signal Handler Install	lat_sig install	1.36	1.52	12.06
Signal Handler Dispatch	lat_sig handle	3.00	4.44	48.18

Table 2.6: High Level Kernel Operations. Times are in μs.

Buffer Size (KB)	**File Native (MB/s)**	**File SVA (MB/s)**	**Decrease (%)**	**Pipe Native (MB/s)**	**Pipe SVA (MB/s)**	**Decrease (%)**
4	276.27	271.47	1.74	264.17	200.84	23.97
16	179.71	177.70	1.12	206.31	170.48	17.37
64	183.20	182.57	0.34	212.59	177.85	16.34
256	111.71	112.60	-0.80	190.14	163.82	13.84
1024	84.37	86.39	-2.39	133.49	118.68	11.09
4096	86.16	86.37	-0.24	120.28	108.92	9.44

Table 2.7: Read Bandwidth for 4 MB file (bw_file_rd and bw_pipe).

Signal handler dispatch shows the greatest overhead (48%). We know that some of the overhead comes from system call overhead (the re-written dispatch code mentioned previously) and some comes from saving the application's state in kernel memory instead of on the user space stack. We suspect that the greatest amount of overhead, though, comes from saving and restoring the FP State on every signal handler dispatch. If true, we will need to revise the FP State intrinsic design to remove the overhead.

We believe the signal handler install overhead is partially due to the system call latency and partially due to the overhead from copying data from user space to kernel space (on our Linux system, signal() is implemented by a library routine that calls sigaction(), which reads from several user space buffers).

2.5.4 Macrobenchmarks

We ran two application-level benchmarks to evaluate the overall performance impact SVA-OS has on two types of common applications: web servers and compilation.

# Clients	Native	SVA	% Decrease
1	81.21	76.16	6.22
4	72.21	68.76	4.78

Table 2.8: Web Server Bandwidth Measured in Megabits/second.

First, we ran the WebStone benchmark [103] on the thttpd web server [115] (Table 2.8). WebStone measures the bandwidth that a web server can provide to a variable number of clients. Our test retrieves files of size 0.5KB to 5MB. The thttpd web server uses a single process and thread to handle I/O requests from web clients [115].

The SVA kernel decreases the bandwidth of the server by less than 7%. We are still trying to determine the cause of the decrease; we have, however, determined that data copying overhead is not the primary cause, unlike an earlier version of the system that had higher overheads [104].

We also benchmarked a build of OpenSSH 4.2p1 [117] using the time command. A complete build took 176.78 seconds on the native kernel and 178.56 seconds on the SVA kernel, yielding a negligible 1.01% overhead. The times are elapsed time (the SVA kernel does not correctly report per process user or system time). The low overhead is because compilation is primarily a user-level, CPU-bound task.

2.6 Related Work

There are several classes of related work on virtualizing operating system code, including (a) OS support in virtualized processor architectures (or "codesigned virtual machines" [131]); (b) hardware abstractions in previous operating systems; (c) virtual machine monitors; and (d) operating systems that exploit language-based virtual machines or safe languages.

Four previous system architectures have used an organization with separate virtual and native instruction sets: the Transmeta Crusoe and its successors [55], the IBM DAISY project [61], the IBM S/38 and AS/400 families of systems [40], and the Low-Level Virtual Architecture (LLVA) [13]. Both Transmeta and DAISY emulated an existing "legacy" hardware ISA (x86 and PowerPC, respectively) as their virtual instruction set on a completely hidden VLIW processor. Both allowed existing operating systems written for the legacy instruction sets to be run with virtually no changes. These systems aim to implement legacy instruction sets

on novel hardware but do not make it easier to analyze or transform operating system code and, therefore, do not make it easier to enforce security policies on operating system or application code.

The IBM AS/400 (building on early ideas in the S/38) defined a high-level, hardware-independent interface called the Machine Interface (MI) that was the sole interface for all application software and for much of OS/400. The major difference between SVA and their MI is that their MI was effectively a part of the operating system (OS/400); significant components of OS/400 ran below the MI and were required to implement it. In their approach, a particular OS is organized to enable many or all of the benefits that SVA provides. SVA is OS-neutral and aims to provide similar benefits to *existing* operating systems without a major reorganization of the OS.

LLVA [13] forms the basis of SVA; it provides a virtual instruction set that is amenable to compiler analysis and transformation and supports C/C++ code. However, LLVA lacks the instructions needed for operating system kernel operations such as context switching, signal handler dispatch, and thread creation. It also provides no safety guarantees for application or kernel code. SVA extends LLVA with these features.

Many modern operating systems include some design features to separate machine-independent and machine-dependent code, and at least a few do this by using an explicit architecture abstraction layer to define an interface to the hardware. Two such examples include the Windows Hardware Abstraction Layer (HAL) [119] and the Choices nanokernel [32]. These layers are integral parts of the OS and only achieve greater machine-independence and portability. In contrast, SVA's instructions are an integral part of the (virtual) processor architecture and yet provides richer primitives than a traditional architecture for supporting an OS.

The Alpha processor's PALCode layer [41] provides an abstraction layer similar to the one we designed for SVA. PALCode is special privileged code running below the OS that can execute special instructions and access special registers, e.g. the TLB. PALCode routines, like SVA instructions, act like additional instructions in the processor's instruction set, have privileged access to hardware, and provide a higher level interface to the processor compared with traditional instruction sets. PALCode's goals, however, differ significantly from SVA's. PALCode is co-designed with the OS and moves *OS specific* functions into the abstraction layer. It does not hide the processor's native ISA from the OS. Because it is OS-defined, it is unrecognizable by external compilers. In contrast, SVA hides the entire native ISA from software, provides OS-neutral abstractions, and has well-defined semantics. PALCode is logically part of the OS, whereas SVA is part of the processor architecture.

Virtual machine monitors such as VMWare [147], Denali [152] and Xen [60] virtualize hardware resources to enable sharing and isolation for multiple instances of operating systems. These systems are orthogonal to our work, which virtualizes the instruction set interface for a single instance of an OS. That said, standardized interfaces for paravirtualized hypervisors [18] (in which the operating system kernel is modified for virtualization) abstract the hardware in a way similar to SVA, and Linux's `paravirtops` interface [155] was the basis of SVA's MMU configuration instructions (see Chapter 5).

Systems such as JavaOS [122], J-Kernel [74], JRes [53], KaffeOS [27] and the Java extension defined in JSR-121 [79] have incorporated OS-like features into or as a layer on top of a JVM. These systems aim to provide OS services to Java applications but are not designed to be general-purpose operating systems.

Chapter 3

Secure Virtual Architecture

3.1 Introduction

Many security policies are enforced via compiler instrumentation; examples include control-flow integrity [10], memory safety [108, 57, 15, 16], and information flow [85]. Furthermore, the protection provided by such compiler instrumentation can often be improved with the use of accurate static analysis. For example, SAFECode [57] and WIT [15] provide stronger security guarantees when they use increasingly accurate points-to analysis results. Likewise, Data-Flow Integrity (DFI) [33] provides stronger security when it can compute increasingly accurate results for reaching definitions analysis.

In this chapter[1], we further describe the *Secure Virtual Architecture* (SVA) from Chapter 2. SVA is designed to support modern operating systems efficiently and with relatively little change to the OS. SVA leverages the virtual instruction set provided by LLVA to accurately and statically analyze OS kernel code and to instrument kernel code with run-time checks to enforce security policies. Since the OS kernel can be expressed without the use of assembly language code, SVA avoids the complications of analyzing and instrumenting assembly code. In addition, SVA utilizes the SVA-OS instructions from Chapter 2 to identify when the OS kernel is interacting with the hardware or manipulating application state; this allows SVA to enforce policies that require understanding when these operations take place.

Porting a kernel to SVA requires no major design changes; it is similar to, but significantly simpler than, porting to a new hardware architecture because SVA provides simple abstractions for the privileged operations performed by a kernel. Furthermore, the compilation and execution process is largely transparent: most application and kernel programmers would see no change when running SVA code (unless they inspect object code).

[1] This chapter is derived from a conference paper [50]. Andrew Lenharth helped with the design. The copyright owner has given permission to reprint.

The rest of this chapter is organized as follows. Section 3.2 gives an overview of the SVA design. Section 3.3 describes the virtual instruction set and the kernel boot and execution process. Section 3.4 briefly describes the two implementations of SVA used in this work. Section 3.5 concludes.

3.2 Overview of the SVA Approach

The overall goal of the SVA project is to enforce security policies on commodity operating systems and the security-critical programs that run on them. A unique feature of SVA is that its design permits the use of compiler, virtual machine, and hardware techniques (and a combination thereof) to enforce security policies. We aim to enforce security policies under two major constraints: (a) have only a small impact on performance, and (b) require small porting effort and few OS design changes on the part of OS developers.

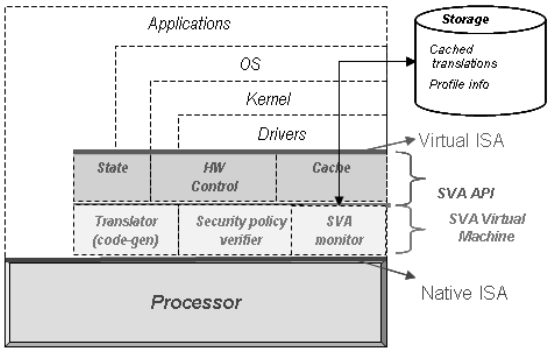


Figure 3.1: System Organization with SVA

Figure 3.1 shows the system organization with SVA. Briefly, SVA defines a virtual, low-level, typed instruction set suitable for executing *all* code on a system, including kernel and application code. Part of the instruction set is a set of operations (collectively called SVA-OS, introduced in Chapter 2) that encapsulates all the privileged hardware operations needed by a kernel. The system implementing the virtual instruction set instruments executable "*bytecode*" (i.e., virtual object code) files to ensure that they satisfy the desired security policies. Such instrumentation can be done ahead-of-time or can be done when the bytecode is translated to native code for execution. We refer to this system as a Secure Virtual Machine (SVM) and the virtual object code that it executes as "bytecode."

To further reduce the TCB, an implementation of SVA can employ a separate "verifier" component to verify that the instrumentation added to enforce a security policy is correct. As will be seen in Chapter 4, a

verifier is often simpler than the static analysis used to strengthen and optimize run-time checks. Employing a verifier removes the need to trust complicated static analysis components to enforce the security policy and thereby reduces the size of the TCB.

Porting an existing kernel to SVA involves three steps. First, the target-dependent layer of the kernel is ported to use the SVA-OS interface. This is similar to porting to a new architecture but potentially simpler because the SVA-OS operations are slightly higher-level and more abstract. At the end of this step, there will be *no explicit assembly code* in the kernel. Second, parts of the kernel not conforming to the security policy enforced by SVA must be modified to conform. For the policies explored in this work, this is almost never needed; the only exception is a small change to the Linux kernel memory allocators for memory safety described in Chapter 4. Finally, some optional improvements to kernel code can significantly improve static analysis precision and hence the security guarantees provided by SVA.

The steps by which kernel or secure application code is compiled and executed on SVA are as follows. A front-end compiler translates source code to SVA bytecode. This code is then shipped to end-user systems. At install time or load time, the SVM either checks the security policy enforcement with the *bytecode verifier* or instruments the bytecode on the end-user system to enforce the desired security policy. A *translator* transparently converts the bytecode into native code for a particular processor. The instrumentation, verification, and translation process can happen offline (to maximize performance) or online (to enable additional functionality). When translation is done offline, the translated native code is cached on disk together with the bytecode, and the pair is digitally signed *together* to ensure integrity and safety of the native code. In either case, kernel modules and device drivers can be dynamically loaded and unloaded (if the kernel supports it) since they can either be checked to verify that they enforce the security policy or can be instrumented on-demand to enforce the security policy. If needed, native code modules can be dynamically loaded even if they were not translated to native code by the SVM; such external "unknown" code must be treated as trusted and is essentially part of the TCB.

An important issue that we do not address in this work is kernel recovery after a violation of the security policy is detected at run-time. Several systems, including Vino [124], Nooks [140, 142], SafeDrive [164], and Lenharth et. al.'s recovery domains [89], provide strategies for recovering a commodity kernel from a fault due to a device driver or other extension. SVA does not yet include any specific mechanisms for recovery, and investigating recovery mechanisms (including these existing techniques, which should be compatible with SVA) is a subject of future work.

3.3 The SVA Execution Strategy

The SVA virtual instruction set has two parts: the core computational instructions (SVA-Core) which are used by all software running on SVA for computation, control flow, and memory access, and the OS support operations (SVA-OS). We now describe the execution strategy for a kernel running on the SVM.

3.3.1 Instruction Set Characteristics

The SVA-Core instructions and the SVA object file format are inherited directly from the LLVM compiler infrastructure [86]. LLVM (and hence, SVM) defines a single, compact, typed instruction set that is used both as the in-memory compiler internal representation (IR) and the external, on-disk "bytecode" representation. This is a simple, RISC-like, load-store instruction set on which detailed optimizations can be expressed directly, like a machine language, but unlike a machine language, it also enables sophisticated analysis and transformation. There are four major differences from native hardware instruction sets. First, memory is partitioned into code (a set of functions), globals, stack, and other memory. Second, every function has an explicit control flow graph with no computed branches. Third, the instruction set uses an "infinite" virtual register set in Static Single Assignment (SSA) form, making many dataflow analyses simple and powerful. Fourth, the instruction set is typed, and all instructions are type-checked. The type system enables advanced techniques for pointer analysis and array dependence analysis. Unsafe languages like C and C++ are supported via explicit cast instructions, a detailed low-level memory model, and explicit control over memory allocation and deallocation.

An SVA object file (called a *Module*) includes functions, global variables, type and external function declarations, and symbol table entries. Because this code representation can be analyzed and transformed directly, it simplifies the use of sophisticated compiler techniques at compile-time, link-time, load-time, run-time, or "idle-time" between program runs [86]. In fact, both the security policy instrumentation code and the bytecode verifier within the SVM operate on the same code representation. Furthermore, because the bytecode language is also designed for efficient just-in-time code generation, the bytecode verification and translation steps can easily be done at load-time for dynamically loaded modules.

3.3.2 The SVA Boot and Execution Strategy

The Secure Virtual Machine (SVM) implements SVA by performing bytecode instrumentation and translation and implementing the SVA-OS instructions on a particular hardware architecture. An implementation can

optionally implement native code caching and authentication and bytecode verification. Section 3.2 briefly discussed the high level steps by which an SVA kernel is loaded and executed. Here, we describe some additional details of the process.

On system boot, a native code boot loader will load both the SVM and OS kernel bytecode and then transfer control to the SVM. The SVM will then begin execution of the operating system, either by interpreting its bytecode or translating its bytecode to native code. For handling application code during normal operation, the SVM can use callbacks into the operating system to retrieve cached native code translations. These translations can be cryptographically signed to ensure that they have not been modified.

Since the SVM translates all code on the system (including the OS kernel), it is able to exercise a great degree of control over software execution. It can inline reference monitors [63] during code generation; it can choose to execute software in less privileged processor modes (similar to hypervisors like Xen [60]). Using either or both of these techniques allows the SVM to mediate access to privileged processor resources (like the MMU) and to enforce a wide range of policies.

The SVM uses two methods to obtain memory for its own execution. First, it reserves a portion of physical memory for its initial use ("bootstrap"); this memory is used for code and internal data needed during system boot before the OS kernel has become active. The amount of memory needed for bootstrap is fixed and statically determinable; the version of SVA that enforces memory safety (Chapter 4) reserves about 20KB. Second, during normal system operation, the SVM uses callback functions provided by the OS to obtain memory from and release memory to the OS. Since the SVM mediates all memory mappings, it can ensure that the memory pages given to it by the OS kernel are not accessible from the kernel or any other program on the system.

3.4 Implementations

We implemented two versions of SVA. The first version is for 32-bit x86 processors and is based on the LLVA implementation from Chapter 2. While this version can run on multi-processor hardware, it only supports single-core, single processor execution. As described in Chapter 2, we ported Linux 2.4.22 to this version of SVA.

We also implemented a second version of SVA for 64-bit x86 systems. This version only supports 64-bit code (even though the processor supports 32-bit code). We wrote this version of SVA so that it could support multi-core, multi-processor systems. However, since we have not devised a suitable locking discipline for concurrent updates to page tables in the MMU instructions, this new implementation only operates in a

single-core, single processor mode. We ported FreeBSD 9.0 to this new version of SVA because FreeBSD 9.0 compiles with LLVM "out of the box" whereas the Linux did not when we started the project.

3.5 Summary

Compiler instrumentation is a common technique for enforcing security policies. However, the design of current commodity operating systems makes the application of compiler techniques difficult: a compiler must reason about hand-written assembly code, and it has no good way of inferring which OS kernel operations manipulate privileged hardware or application state.

The Secure Virtual Architecture (SVA) leverages virtual instruction set computing to overcome these challenges, thereby enabling the use of compiler instrumentation to enforce security policies on commodity OS kernel code. As Chapters 4, 5, and 6 will show, strong security policies such as memory safety and control-flow integrity can be enforced with SVA. Using its control over OS/application interactions, SVA can even be used to protect applications from a compromised OS kernel (as Chapter 7 demonstrates).

Chapter 4

Memory Safety for a Commodity Operating System Kernel

4.1 Introduction

Despite many advances in system security, most major operating systems remain plagued by security vulnerabilities. One major class of such vulnerabilities is *memory safety errors*, such as buffer overruns [17], double frees (e.g., [59, 2, 3]), and format string errors [127]. Rapidly spreading malware like Internet worms, especially those that use "self-activation" [150], exploit such errors because such attacks can be carried out in large numbers of systems quickly and fully automatically.

Safe programming languages, such as Java and C#, guarantee that such errors do not occur, but "commodity" operating systems like Windows, Mac OS X, Linux, FreeBSD, and Solaris (as well as security-sensitive software such as OpenSSH and Apache) are all written using C and C++, and hence enjoy none of the safety guarantees of safe languages.

There have been several previous approaches for enforcing various safety guarantees for commodity OS code. On the one hand, approaches like Software Fault Isolation [148] or XFI [144] enforce isolation between coarse-grained components but do not provide "fine-grained safety guarantees" (e.g., for individual buffers, pointers, or heap objects) needed for a safe execution environment. Similarly, HyperSafe [149] only enforces control-flow integrity and does not protect individual memory objects from corruption. At the other extreme, language-based approaches such as Cyclone [71, 29] and SafeDrive [164] provide a safe execution environment but have only been applied to specific components, appear difficult to extend to a complete kernel, and require significant changes or annotations to existing kernel code.

In this chapter[1], we describe a version of SVA, dubbed SVA-M, and its implementation, designed both to support modern operating systems efficiently and with relatively little change to the guest OS. SVA-M provides a safe execution environment for a kernel and (selected) application code.

[1]This chapter is derived from a conference paper [50]. Andrew Lenharth helped with the design, helped run some of the performance experiments, and ran the experiments on the analysis results. Dinakar Dhurjati built and evaluated the type checker. The copyright owner has given permission to reprint.

The safety guarantees SVA-M provides, listed in detail in Section 4.3.9, include *memory safety*, *control-flow integrity*, *type safety for a subset of objects*, and support for *sound program analysis*. These guarantees are close to, but slightly weaker than, the safety guarantees provided by a safe language like Java, C#, or Modula-3. There are two essential weaknesses which occur because SVA-M preserves the low-level memory model of C: (a) dangling pointer references can occur but are rendered harmless, i.e., they cannot violate the safety guarantees (although they may still represent potential logical errors); and (b) arbitrary, explicit casts are permitted, as in C (in fact, we support the full generality of C code with no required source changes). These compromises allow us to achieve a fairly strong degree of safety for commodity OSs while minimizing kernel porting effort.

SVA-M is also designed to ensure that the (relatively complex) static analysis and compiler instrumentation components do not need to be a part of the Trusted Computing Base (TCB). The compiler component can generate code in the SVA virtual instruction set, and a simple type checker can ensure that the code meets the safety requirements of SVA-M. This provides a *robust* implementation strategy for enforcing the safety properties of SVA-M.

The rest of this chapter is organized as follows. Section 4.2 gives an overview of SVA-M's design. Section 4.3 explains the approach to enforcing safety in SVA-M. Section 4.4 explains the SVA-M type system and type checker which help minimize the SVA-M TCB. Section 4.5 describes how we ported the Linux kernel to SVA-M. Section 4.6 presents our experimental evaluation, including performance overheads, effectiveness at catching previously reported kernel exploits, and static metrics on the effectiveness of the memory safety instrumentation. Finally, Section 4.7 compares SVA-M with previous work, and Section 4.8 concludes.

4.2 Overview of SVA-M

The broad goals of the SVA-M design are to provide a safe execution environment for commodity operating systems and the security-critical programs that run on them. We aim to achieve this goal while having only a small impact on performance while requiring a small porting effort and few OS design changes. To achieve these goals, we utilize the SVA infrastructure described in Chapter 3.

Figure 4.1 shows the SVA-M system organization. SVA-M contains components for instrumenting SVA bytecode with run-time checks to enforce memory safety and type safety and translating SVA bytecode to native code. It also contains a bytecode verifier that verifies that the instrumentation added by the compiler instrumentation component is correct. SVA-M also extends the SVA instruction set and its type system to encode its memory safety and type safety security policies. These policies extend the safety strategy

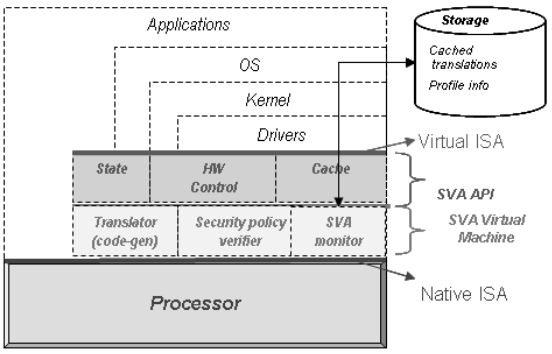


Figure 4.1: SVA-M System Organization

developed in the SAFECode memory safety compiler [57, 56] to work in a full-fledged commodity kernel as Section 4.3 describes.

4.3 Enforcing Safety for Kernel Code

In this section, we first give some brief background on SAFECode, the previous work on enforcing fine-grained safety properties for stand-alone C programs (SVA-M uses SAFECode directly to provide a safe execution environment for applications). We then identify three major challenges that arise when applying the safety principles in SAFECode to a commodity kernel. In the subsequent sections, we present our solutions to these challenges and then summarize concretely the specific safety guarantees SVA-M provides for kernel code.

4.3.1 Background: How SAFECode Enforces Safety for C Programs

The SAFECode compiler and run-time system together enforce the following safety properties for a *complete, standalone C program with no manufactured addresses* [57, 58, 56]:

(T1) *Control-flow integrity*: A program will never execute an instruction sequence that violates the compiler-computed control flow graphs and call graph.

(T2) *Type safety for a subset of objects*: All objects in type-homogeneous partitions (defined below) are accessed or indexed according to their compiler-inferred type, or arrays of that type.

(T3) *Array bounds safety*: All accesses to array objects (including strings) fall within the array bounds.

(T4) *No uninitialized pointer dereferences*: All successful loads, stores, or calls use correctly initialized pointers.

(T5) *No double or illegal frees*: Dynamic deallocation operations will only be performed for *live* (not previously deallocated) objects with a legal pointer to the start of the allocated object.

(T6) *Sound analysis*: A sound operational semantics [57] is enforced, incorporating a flow-insensitive points-to graph, call graph, and a subset of type information, usable by compilers and static analysis tools to implement *sound* higher-level analyses.

Note that dangling pointer *dereferences* (i.e., *read or write after free*) are *not* prevented. Nevertheless, SAFECode ensures that the other guarantees are not violated.

SAFECode enforces these properties through a combination of program analysis and run-time checks with *no* changes to source code of programs and without preventing the explicit deallocation of objects. Briefly, the SAFECode principles are as follows.

The compiler is given (or computes) a call graph, a "points-to graph" representing a static *partition* of all the memory objects in the available part of the program [88], and type information for a subset of the partitions as explained below. It automatically transforms the program (using an algorithm called Automatic Pool Allocation [87]) so that memory objects in distinct partitions (nodes in the points-to graph) are allocated in different logical "pools" of memory. Individual object allocation and deallocation operations occur at exactly the same locations as in the original program but use the appropriate pool. Stack and global objects are registered with the pool to which they were assigned by the pointer analysis, and stack objects are deregistered when returning from the parent function. A key property SAFECode exploits is that, with a suitable pointer analysis, many partitions (and therefore the corresponding run-time pools) are *type-homogeneous* (TH), i.e., all objects allocated in the pool are of a single (known) type or are arrays of that type.

The fundamental sources of memory safety violations in C programs include *uninitialized variable references*, *array bounds violations* (including format string errors), *dangling pointer references*, and a variety of *illegal type casts* (including improper arguments to function calls). SAFECode prevents these violations and enforces the guarantees above as follows:

- It prevents *uninitialized variable references* via dataflow analysis (for local variables) and via initialization of allocated memory to ensure a hardware-detected fault on dereferences (for all other pointer variables).

- It prevents *array bounds violations* using an extension [56] of the Jones-Kelly approach for detecting array bounds violations [81]. This extension uses a separate run-time search tree (a splay tree) in each pool to record all array objects at run-time and looks up pointer values in this table to check for bounds violations. This strategy allows SAFECode to avoid using "fat pointers" for tracking array bounds at run-time; fat pointers, when used inside structures or variables that are accessible from library functions, are known to cause significant compatibility problems [111, 80, 25].

- Simple compile-time type-checking is sufficient to ensure *type safety of objects in type-homogeneous pools*. Dangling pointers to these objects cannot compromise type safety *as long as the run-time allocator does not release memory of one pool to be used by any other pool, until the first pool is "dead;"* in practice, SAFECode releases memory of a pool only when the pool is unreachable [58]. For pointers to objects in non-TH pools, run-time "pool bounds checks" at dereference ensure the pointer target lies within the pool.[2] Since stack memory can be used to allocate objects of any type (and is therefore not type-homogeneous), stack-allocated objects that may escape their function and (except for potential dangling pointers) are type safe are allocated on the heap in a type-homogeneous pool. In this way, these stack objects are guaranteed to be type-homogeneous just like type safe heap objects.

- SAFECode enforces *control-flow integrity* by generating native code itself (preventing illegal branches to data areas), preventing writes to code pages, and using run-time checks to ensure that indirect function calls match the call targets computed by the compiler's call graph analysis.

- Finally, the *soundness of the operational semantics*, which requires correctness of the given analysis information (i.e., the call graph, points-to graph, and type information for TH partitions) follows directly from the previous safety properties [57].

Partitioning memory into logical pools corresponding to the pointer analysis has several critical benefits for the SAFECode approach:

[2] We could instead perform more precise "object bounds" checks, using the same search trees as for array bounds violations. That is what we do in SVA-M.

- Type-homogeneity directly gives type safety for some objects.
- Type-homogeneous pools and run-time checks for non-TH pools together make dangling pointers harmless.
- No run-time checks are needed on dereferences of a pointer between TH pools.
- Using a separate splay tree per pool and eliminating scalar objects from the splay tree in TH pools make array bounds checking orders-of-magnitude faster than the original Jones-Kelly method. In fact, it is enough to make this approach practical [56].

Note that the SAFECode guarantees are weaker than a safe language like Java or C#. In particular, SAFECode does not prevent or detect dangling pointers to freed memory (safe languages prevent these by using automatic memory management) and permits flexible pointer casts at the cost of type safety for a subset of memory. The SAFECode approach provides a useful middle ground between completely safe languages and unsafe ones like C/C++ as it does not impose much performance overhead, does not require automatic memory management, is able to detect all types of memory safety errors other than dangling pointer uses, and is applicable to the large amount of legacy C/C++ code.

4.3.2 SAFECode for a Kernel: Challenges

We use the SAFECode compiler directly in SVA-M for standalone programs. Extending the SAFECode safety principles from standalone programs to a commodity kernel, however, raises several major challenges:

- The custom allocators used by kernels to manage both correctness and performance during memory allocation are incompatible with the pool allocation strategy that is fundamental to SAFECode.
- Unlike standalone programs, a kernel contains many entry and exit points from or to external code, and many of these bring pointers into and out of the kernel.
- Unlike most applications, a kernel typically uses a number of "manufactured addresses," i.e., where a predefined integer value is used as an address, and these can be difficult to distinguish from illegal addresses.

In the following subsections, we present our solution to these challenges. Preserving custom allocators is, by far, the most difficult issue and is a major focus of this section.

4.3.3 Integrating Safety Checking with Kernel Allocators

The SAFECode approach critically depends on a specific custom allocation strategy, namely, pool allocation with pools partitioned according to a pointer analysis. Kernels, however, make extensive use of custom allocators, e.g., `_alloc_bootmem`, `kmem_cache_alloc`, `kmalloc` and `vmalloc` in Linux [30], or the `zalloc`, `kalloc` and `IOMalloc` families of allocators in the Darwin kernel of Mac OS X [130]. Running SAFECode directly on a kernel (even by identifying the kernel allocators to the compiler) is impractical because the compiler-generated allocation strategy would not work: the kernel allocators ensure many complex *correctness* requirements, e.g., pinning memory, alignment requirements, virtual memory mappings, and other issues.

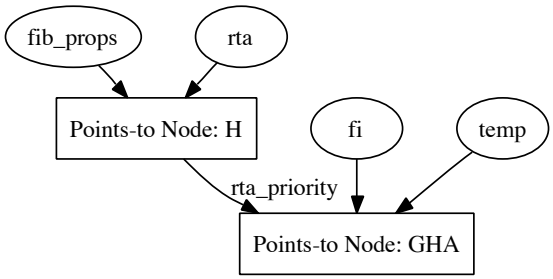
The key insight underlying our solution is that *a kernel typically already uses pool allocation and furthermore, many of the "pools" are type-homogeneous*: distinct pools are created for different kinds of heavily used kernel objects. For example, a Darwin reference gives a partial list of 27 different data types (or classes of data types) for which a separate "zone" per data type is created using the zone allocator. In Linux, at least 37 different "caches" are created using the `kmem_cache_create` operation (not including the non-type-homogeneous caches used within the more generic `kmalloc` allocator). This suggests that if we can map existing kernel pools with pointer-analysis partitions, we may be able to achieve the safety benefits we seek without changing the kernel's custom allocators. We do so as follows.

First, as part of the porting process, kernel developers must identify the allocation routines to the compiler and specify which ones should be considered "pool allocators;" the rest are treated as ordinary allocators. The existing interface to both kinds of allocators are not modified. For each allocator, routines must be specified for object allocation and deallocation and, for pool allocators, the routines for creation, initialization, and destruction. The specific requirements for porting kernel allocators are described in Section 4.3.4.

The primary goal of the analysis, then, is to correlate kernel pools and other kernel objects with the static partitions of memory objects (i.e., nodes in the points-to graph) computed by pointer analysis. The compiler does *not* use the Automatic Pool Allocation transformation to partition memory into pools.

The output of pointer analysis is a points-to graph in which each node represents a distinct partition of objects in the analyzed part of the program [88]. An assumption in both SAFECode and SVA-M is that the pointer analysis is a "unification-style" algorithm [139], which essentially implies that every pointer variable in the program points to a unique node in the points-to graph. A single node may represent objects of multiple memory classes including *Heap*, *Stack*, *Global*, *Function* or *Unknown*. Figure 4.2 shows a part of a points-to graph for a code fragment from the Linux kernel. The `G` and `H` flags on the node pointed to by `fi`

```
1  MetaPool MP1, MP2;
2
3  struct fib_info * fib_create_info(
4    const struct rtmsg *r, struct kern_rta *rta,
5    const struct nlmsghdr *nlh, int *errp) {
6  ...
7    //look up object bounds and then check the access
8    getBounds(MP1, &fib_props, &s, &e);
9    boundscheck(s, &fib_props[r->rtm_type].scope, e)
10   if (fib_props[r->rtm_type].scope > r->rtm_scope)
11      goto err_inval;
12 ...
13   fi = kmalloc(sizeof(*fi)+nhs*sizeof(struct
14                fib_nh), GFP_KERNEL);
15   pchk_reg_obj(MP2, fi, 96, NULL, SVA_KMALLOC);
16 ...
17   //check bounds for memset without lookup since
18   //we know the start and size from the kmalloc
19   boundscheck(fi, (char*)fi + 95, (char*)fi + 96);
20   memset(fi,0,sizeof(*fi)+nhs*sizeof(...));
21
22   //check that rta is a valid object
23   lscheck(MP1, rta);
24   if (rta->rta_priority) {
25      //check that rta->rta_priority is valid
26      temp = rta->priority;
27      lscheck(MP2, temp);
28      fi->fib_priority = *temp;
29   }
30 ...
31 }
```



Example code fragment from the Linux kernel, and a part of the points-to graph for it. Square boxes are nodes representing memory objects. Ovals represent virtual registers.

Figure 4.2: Points-to Graph Example

indicate that the node includes global objects as well as heap objects such as the one allocated with `kmalloc` at line 13.

Given the points-to graph, the compiler creates a global *metapool* variable for each node in the graph, e.g., `MP1` and `MP2` in the example. A metapool is simply a set of data objects that map to the same points-to node and so must be treated as one logical pool by the memory safety instrumentation algorithm. Metapool variables, like the pool descriptors in SAFECode [57], are used as run-time representations of each points-to graph partition, recording run-time information ("metadata") about objects to enable run-time checks. They are also represented as types on pointer variables to be checked by the bytecode verifier. Using a global variable for each metapool avoids the need to pass metapool variables between functions.

Name	Description
`pchk.reg.obj` (MetaPool * MP, void * address, unsigned length, void * pool, int alloctrID)	Register an object starting at `address` of `length` bytes with the MetaPool `MP`. The `pool` and `alloctrID` arguments track which allocator and kernel pool (if applicable) were used to allocate the object.
`pchk.drop.obj` (MetaPool * MP, void * address, void * pool, int alloctrID)	Remove the object starting at `address` from the MetaPool `MP`. If `MP` is complete, perform an invalid free check.

Table 4.1: Instructions for Registering Memory Allocations

At every heap allocation point, identified using the kernel allocator functions specified during porting, we insert a `pchk.reg.obj` operation to register the allocated object in the appropriate metapool (the one mapped to the pointer variable in which the allocation result is stored), e.g., at line 15 of Figure 4.2. Similarly, the compiler inserts a `pchk.drop.obj` operation at every deallocation point. These two operation signatures are shown in Table 4.1.

The compiler also inserts `pchk.reg.obj` operations to register all global and stack-allocated objects, and `pchk.drop.obj` to drop stack objects at function exits. Global object registrations are inserted in the kernel "entry" function where control first enters during the boot process. Stack-allocated objects that may have reachable pointers after the parent function returns (which can be identified directly from the points-to graph) are converted to be heap allocated using a kernel-specified allocator and deallocated at function return. This tolerates dangling pointers in the same way as heap objects: by controlling the reuse within TH pools and performing run-time checks within non-TH pools.

One key requirement for achieving our guarantees is that all memory managed as a single pool of memory (i.e., with internal reuse) must be registered in a single metapool: if it was spread across multiple metapools, a dangling pointer from one metapool could point to a different metapool, which could violate both type-

safety (if one metapool is TH and any of the other metapools is of a different type) and sound pointer analysis (since each metapool represents a distinct node in the points-to graph). If a single kernel pool (e.g., a single `kmem_cache_t` object in Linux) maps to two or more partitions, we merge the points-to graph nodes for those partitions into a single graph node (effectively making the points-to analysis less precise, but still correct). Note that the converse – multiple kernel pools in a single metapool – needs no special handling for achieving our guarantees. (We only need to deregister all "remaining" objects that are in a kernel pool when a pool is destroyed.)

For the same reason, for ordinary allocators (e.g., `kmalloc` in Linux), all the memory managed by the allocator has to be treated as a single metapool because it may have full internal reuse. We effectively must merge all metapools (and hence all points-to graph nodes) that represent objects with a particular ordinary allocator. In some cases, however, an ordinary allocator is internally implemented as a default version of a pool allocator, e.g., `kmalloc` internally just uses `kmem_cache_alloc`. By exposing that relationship, as explained in Section 4.5, the kernel developer can reduce the need for unnecessary merging.

At this point, the memory safety instrumentation compiler pass has successfully mapped kernel pools and all pointer variables (including pointers to globals and stack allocated objects) to metapool variables. The compiler finally encodes the list of metapools and these mappings as type attributes on the SVA bytecode. This bytecode will later be presented to the verifier which type-checks the program (as described in Section 4.4). The specific ways in which we applied the overall strategy above to the Linux kernel is described in Section 4.5.

4.3.4 Kernel Allocator Changes

The kernel's allocators must fulfill several requirements in order to support the run-time checks needed for our memory safety strategy:

- The kernel source code must identify which allocators can be treated as pool allocators and declare the allocation and deallocation functions used for each distinct allocator. The kernel must also provide an ordinary (non-pool) allocation interface that is available *throughout* a kernel's lifetime for stack-to-heap promotion. Internally, this interface could be implemented to use distinct allocators at different times (e.g., during boot vs. normal operation).

- Each allocator must provide a function that returns the size of an allocation given the arguments to the allocation function. This allows the compiler to insert `pchk.reg.obj` operations with the correct size.

- A type-homogeneous pool allocator must allocate objects aligned at the type size (or integer multiples thereof) to ensure that references using a dangling pointer do not cause a type conversion when accessing a newly allocated object.
- A kernel pool allocator must not release freed memory back for use by other pools, though it can reuse memory internally (technically, it can also release memory to other pools within the same metapool, though we do not currently provide such a mechanism).
- Kernel allocators must initialize memory to zero on each allocation (the first page of virtual memory should be unmapped to generate a fault if accessed).

Except for the third and fourth restrictions – on object alignment and memory release – there are no other changes to the allocation strategy or to the internal metadata used by the allocator. Note that these porting requirements are difficult to verify, i.e., SVA-M trusts the kernel developer to perform these changes correctly. However, performing all object registration and deregistration under compiler control and avoiding adding any metadata to the kernel allocators reduces the level of trust placed on the kernel developer.

4.3.5 Run-time Checks

The compiler instrumentation component of SVA-M is responsible for inserting run-time checks into the kernel bytecode. The run-time checks work as follows.

Each metapool maintains a splay tree to record the ranges of all registered objects. The run-time checks use these splay trees to identify legal objects and their bounds. The run-time checks SVA-M performs are:

1. *Bounds Check*: A bounds check must be performed for any array indexing operation that cannot be proven safe at compile-time, e.g., the checks at lines 9 and 19 in Figure 4.2. In the SVA-M instruction set, all indexing calculations are performed by the `getelementptr` instruction which takes a source pointer and a list of indexes and calculates a new address based upon the index list and the source pointer's type. A `boundscheck` operation verifies that the source and destination pointer belong to the same object within the correct metapool.

   If SVA-M can determine the bounds expressions for the target object of the source pointer, those bounds can be used directly, as at line 19 in the example. Otherwise, SVA-M must insert a `getBounds` operation to verify that the object is in the correct metapool and then use the fetched bounds of that object for the check.

Bounds checks should be performed on structure indexing operations. However, since these operations are usually safe and the performance overheads often outweigh any safety benefits gained, we have chosen not to perform structure indexing checks in our prototype.

2. *Load-store check*: A check must be performed on any load or store through a pointer obtained from a non-type-homogeneous metapool since such pointer values may come from arbitrary type casts instead of through legal, verified pointer operations. The `lscheck` operation is used to verify that the pointer points to a legal object within the correct metapool. Lines 23 and 27 show two examples of such checks.

3. *Indirect Call Check*: An indirect call check verifies that the actual callee is one of the functions predicted by the call graph by checking against the set of such functions. As with load-store checks, this check is not needed if the function pointer is obtained from a type-homogeneous pool because *all* writes to such pools have been detected (including any possible writes through dangling pointers).

4. *Illegal Free Check*: An illegal free check verifies that the argument to a kernel deallocator was allocated by the corresponding kernel allocator. This check is integrated with the `pchk.drop.obj` operation.

One complication is that some partitions may be exposed to external code that is not compiled by SVA-M; the pointer analysis marks these partitions "Incomplete" [88]. Incomplete partitions may include objects not allocated within the available kernel code and, therefore, not registered with the corresponding metapool. This forces SVA-M to use conservative run-time checks. First, load-store, indirect call, and illegal free checks using a pointer to an incomplete partition are useless and must be turned off: even if they detect an address that is not part of the metapool, they cannot tell if it is an illegal address or simply an unregistered, but legal, object or function target. Second, checks on array indexing operations look up the operand and result of the `getelementptr` instruction. If either pointer points to an object in the splay tree, then the check can be performed, failing if both pointers are not within the same object. If the target object is not in the splay tree, nothing can be said. Overall, this means that "incomplete" partitions only have bounds checks on registered objects. We refer to this situation as "*reduced checks*." Reduced checks are the sole source of false negatives in SVA-M, i.e., cases where a memory safety violation is not detected.

4.3.6 Multiple Entry Points

A kernel contains many entry points, including system calls, interrupts, and traps. System calls, in particular, bring pointers to objects allocated outside the kernel in via their parameters. SVA-M must ensure safety of kernel objects while still allowing access to these objects even though they are not registered. We observe

that pointer arguments in system calls may have three valid destinations. They may point to userspace objects, they may point to kernel objects when the kernel issues a system call internally, or they may point to objects returned by the kernel to userspace.

Objects in userspace are handled by registering all of userspace as a single object with every metapool reachable from system call arguments. Thus accesses to them are checked, and the checks pass since they find valid objects. This mechanism also ensures that userspace pointers stay in userspace; if an attacker tries to pass a buffer that starts in userspace but ends in kernel space in an attempt to read kernel memory, this will be detected as a bounds violation.

Internal system calls, those issued from within kernel space, are analyzed like any other function call; thus, the metapools reachable from the arguments already contain the objects being passed in.

Finally, we simply assume that the the last case – where user programs pass in pointers to kernel objects through system call arguments – does not occur. If a kernel wanted to allow this, SVA-M could support this via minor changes to the pointer analysis to allow checking on these objects. However, this is a very poor design for safety and stability reasons, so it is not supported.

4.3.7 Manufactured Addresses

Unlike most applications, a kernel typically uses a number of "manufactured addresses." These are most often used for accessing BIOS objects that exist on boot at certain addresses. These are, in effect, memory objects which are allocated prior to the start of the kernel. The kernel developer simply must register these objects prior to first use (using the SVA-M function `pseudo_alloc`), which the compiler then replaces with `pchk.reg.obj`, thus treating them like any other allocation. For example, in Linux's ACPI module, we insert `pseudo_alloc(0xE0000, 0xFFFFF)` before a statement that scans this range of memory for a particular byte signature.

There are also C programming idioms that can *appear* to manufacture an address out of integer values even if they do not; most can be handled simply by tracking (pointer-sized) integer values as potential pointers during the pointer analysis, which is generally a necessary requirement for C compilers [68]. Some additional cases, however, may be too expensive to analyze completely, e.g., when bit or byte operations on small integers are used. For example, the Linux kernel performs bit operations on the stack pointer to obtain a pointer to the current task structure. Such operations have the problem that the resulting pointer can point to *any partition* in the points-to graph, complete or incomplete, since the address could be any absolute address.

One solution is simply to have SVA-M reject the code, requiring that such operations be rewritten in a more analyzable form. We take this approach in the current system, modifying the Linux source to eliminate such operations as explained in Section 4.5.3. Alternatively, we could ignore such operations to reduce initial porting effort, essentially trusting that they do not cause any violations of our safety guarantees.

4.3.8 Analysis Improvements

We have added several features to the pointer analysis to improve precision when analyzing kernel code. The Linux kernel often uses small constant values (e.g., 1 and -1) as return values from functions returning a pointer to indicate errors (bugs caused by this have been noted before [62]). These values appear as integer-to-pointer casts and would cause partitions to be marked unknown. We extended the analysis to treat such values (in a pointer context) simply as null. We also added limited tracking of integer dataflow to find all sources of these small constants for a cast site.

Since the kernel may issue system calls internally using the same dispatch mechanism as userspace, namely a trap with the system call (and hence kernel function) specified by numeric identifier, we had to be able to map from syscall number to kernel function in the analysis. This information can be obtained by inspecting all calls to the SVA-OS operation, `sva_register_syscall`, which is used by the kernel to register all system call handlers. With this information, we were able to analyze internal system calls simply as direct calls to the appropriate function.

We use a new heuristic to minimize the extent to which userspace pointers alias kernel memory. The original pointer analysis recognized and handled `memcpy` and `memmove` operations (and their in-kernel equivalents) as copy operations, but simply handled them by merging the source and target object nodes (i.e., handling the copy like `p = q` instead of `*p = *q`). However, for copy operations to or from userspace pointers (which are easily identifiable), we want to minimize the merging of kernel and userspace objects. Our new heuristic merges only the target nodes of outgoing edges of the objects being copied, but it requires precise type information for the source and target objects. If that type information is not available, the analysis *collapses* each node individually (sacrificing type safety) while preventing merging of the nodes themselves.

We also made two improvements that are not specific to kernel code. First, we can reduce spurious merging of objects by judiciously cloning functions. For example, different objects passed into the same function parameter from different call sites appear aliased and are therefore merged into a single partition by the pointer analysis. Cloning the function so that different copies are called for the different call sites eliminates this merging. Of course, cloning must be done carefully to avoid a large code blowup. We used

several heuristics to choose when to create a clone of an existing function. The heuristics have been chosen intuitively and more experience and experiments are needed to tune them. Nevertheless, we saw significant improvements in the points-to graph and some improvements in the type information due to reduced merging, and the total size of the SVA-M bytecode file increased by less than 10%.

Second, the largest source of imprecision in the analysis results comes from certain hard-to-analyze indirect calls, especially through function pointers loaded from global tables attached to objects. Because the tables may mix functions with many different signatures, type safety is completely lost at an indirect call site with such a function pointer, even though the code itself never invokes incompatible functions at the call site. We introduce an annotation mechanism that kernel programmers can use at a call site to assert that the function signatures of all possible callees match the call. In some cases, this can reduce the number of valid targets at an annotated call site by two orders of magnitude. For example, for 7 call sites in Linux, the number of callees went down from 1,189 each (they all get their call targets from the same points-to graph node) to a range of 3-61 callees per call site. This improves analysis precision since fewer behaviors of callees are considered; safety, since fewer control flow paths exist and the programmer is making an assertion about which are valid; and speed of indirect call target checks, since the check is against a much smaller set of possible functions. In fact, with a small enough target set, it is profitable to "devirtualize" the call, i.e., to replace the indirect function call with an explicit `switch` or branch, which also allows the called functions to be inlined if the inliner chooses. The current system only performs devirtualization at the indirect call sites where the function signature assertion was added.

4.3.9 Summary of Safety Guarantees

The SVA-M guarantees provided to a kernel vary for different (compiler-computed) partitions of data, or equivalently, different metapools. The strongest guarantees are for partitions that are proven *type-homogeneous* and *complete*. For partitions that lack one or both of these properties, the guarantees are correspondingly weakened.

Type-homogeneous, complete partitions:

For such partitions, the guarantees SVA-M provides are exactly [T1-T6] listed in Section 4.3.1. Note that the type safety guarantee (T2) applies to all objects in these partitions.

Non-type-homogeneous, complete partitions:

For such partitions, all the above guarantees hold *except*:

(N1) *No type safety*: Memory references may access or index objects in ways inconsistent with their type.

Note that *array indexing* and loads and stores are still checked and enforced; pointer arithmetic or bad casts cannot be used to compute and then use a pointer outside the bounds of an object. This is valuable because it prevents buffer overruns due to common programming errors like incorrect loop bounds, incorrect pointer arithmetic, illegal format strings, and too-small allocations due to integer overflow/underflow. Even if an illegal pointer-type cast or a dangling pointer use converts an arbitrary value to a pointer type, it can only be used to access a legal object within the correct partition for that pointer variable (this is stronger than CCured [111] and SafeDrive [164] which only guarantee that a pointer dereference on a wild pointer will access *some* pointer value, but it can be to an arbitrary object).

Incomplete partitions:

Incomplete partitions must be treated as non-type homogeneous because the analysis for them was fundamentally incomplete: operations on such a partition in unanalyzed code may use a different, incompatible type. In addition, some run-time checks had to be relaxed. The guarantees, therefore, are the same as non-TH partitions above, except:

(I1) *No array bounds safety for external objects*: Array bounds are not enforced on "external" objects, even though the objects logically belong to a partition.

(I2) *Loads and stores to incomplete partitions may access arbitrary memory.* As explained in Section 4.3.5, no *load-store checks* are possible on such partitions.

(I3) *Deallocators may be called with illegal addresses.* No *double/illegal free checks* are done on the partition.

A note on component isolation:

We say a particular kernel component or extension is *isolated* from the rest of the kernel if it cannot perform any illegal writes to (or reads from) objects allocated in the rest of the kernel, i.e., except using legal accesses via function arguments or global variables.

With SVA-M, many partitions do get shared between different kernel components because (for example) objects allocated in one component are explicitly passed to others. SVA-M guarantees component isolation if (a) the component only accesses complete TH partitions (completeness can be achieved by compiling the complete kernel); and (b) dangling pointers are ignored (since SVA-M only guarantees "metapool-level" isolation and not fine-grained object-level isolation on such errors).

Even if these conditions are not met, SVA-M *improves but does not guarantee* isolation of a component from the rest of the kernel: many important memory errors that are common causes of memory corruption (e.g., buffer overruns, uninitialized pointers) cannot occur for kernel objects themselves.

4.4 Minimizing the Trusted Computing Base

The approach described thus far uses pointer analysis results to compute the static partitioning of the memory objects in the kernel. This pointer analysis is interprocedural and relatively complex. (In our SAFECode compiler for standalone programs, Automatic Pool Allocation is also a complex interprocedural transformation.) Any bugs in the implementation of such complex passes may result in undetected security vulnerabilities. For this reason, it is important that such a complex piece of software be kept out of the Trusted Computing Base (TCB) of SVA-M. Furthermore, we would also like to have a formal assurance that the strategy described thus far is sound, i.e., it provides the claimed guarantees such as memory safety, type safety for some subset of objects, control flow integrity, and analysis integrity.

The SAFECode safety approach has been formalized as a type system, and the type system together with an operational semantics encoding the run-time checks have been proved sound for a relevant subset of C [57]. We adapted that type system in the context of SVA-M, using metapools instead of the pool descriptors in SAFECode. The type system allows us to use a simple intraprocedural type checker to check that the complex pointer analysis is correct. Furthermore, it also provides a soundness guarantee on the principles used in our work. An ancillary benefit is that the type system *effectively encodes a sophisticated pointer analysis result directly within the bytecode*, potentially simplifying translation to native code since it does not have to repeat such an analysis.

More specifically, the SVA-M type system [57] essentially encodes each pointer with its associated metapool. For example, for a pointer `int *Q` pointing to a metapool M1, the type system defines the pointer's type to be `int *M1 Q`. If another pointer P has type `int *M2 *M3 P`, then it indicates that `P` points to objects in metapool `M3` which in turn contains pointers that point to objects in metapool `M2`. Effectively, this metapool information has been added as an extension of the underlying SVA-M type system. This encoded information in the form of types is analogous to the encoding of a "proof" in Proof Carrying Code [110]. The proof producer uses the results of pointer analysis and the metapool inference algorithm (described in Section 4.3.3) to infer the type qualifiers `M1`, `M2`, and `M3`. It then encodes this information as annotations on each pointer type.

The typing rules in this type system check that annotated types are never violated. For example, if pointer `int *M1 Q` is assigned a value `*P` where P is of type `int *M2 *M3`, then the typing rules flag this as an error. This will check that type annotations inferred by the proof producer are actually correct. This is essentially the same as checking that objects in `M3` point to objects in `M2` and not objects in `M1`.

The type checker implements the typing rules to check that the "encoded" proof is correct. The type checker is analogous to the proof checker in [110]. Because the typing rules only require local information (in fact, just the operands of each instruction), they are very simple and very fast, both attractive features for use within the virtual machine. Thus only the type checker (and not the complex compiler) is a part of the TCB.

The type checker implementation should be carefully tested because it is part of the TCB. Evaluating its correctness experimentally can also increase our confidence that there are no errors in the manual proof of soundness [57]. For this evaluation, we injected 20 different bugs (5 instances each of 4 different kinds) in the pointer analysis results. The four kinds of bugs were incorrect variable aliasing, incorrect inter-node edges, incorrect claims of type homogeneity, and insufficient merging of points-to graph nodes. The verifier was able to detect all 20 bugs.

4.5 Porting Linux to SVA

Porting an operating system to SVA requires three steps, as noted in Section 4.2: porting to SVA-OS, changes to memory allocators, and optional code changes to improve analysis quality. The specific changes we made to port the Linux 2.4.22 kernel [3] to SVA are described below.

One of our goals was to minimize the changes needed to the architecture-independent portions of a kernel. Table 4.2 summarizes the number of changes that were needed. Column "Total LOC" shows the total number of lines of source code in the original kernel, for each major component. The next three columns show the changes (total number of non-comment lines modified, deleted or added) for the three kinds of porting changes listed in Section 4.2. The last column shows the total number of lines of code changed. As the table shows, the number of changes required are quite small for the improved safety benefits they provide. Section 4.6 suggests some additional kernel changes that can improve the analysis effectiveness, but we expect those additional changes to be on the same order of magnitude as the ones shown here.

4.5.1 Porting to SVA-OS

Linux, like most OSes, uses abstractions for interacting with hardware. Porting the Linux kernel to use SVA was a relatively simple matter of rewriting the architecture-dependent functions and macros to use SVA-OS instructions instead of assembly code, resulting in a kernel with *no inline assembly code*. The total number of architecture-dependent changes from `arch/i386` is shown as the last line in Table 4.2. In some

[3] Linux 2.4.22 was a standard kernel in use when we began this project.

Section	Total LOC	SVA-OS	Allocators	Analysis	Total Modified
Arch-indep core	9,822	41	76	3	120
Net Drivers	399,872	12	0	6	18
Net Protocols	169,832	23	0	29	52
Core Filesystem	18,499	78	0	19	97
Ext3 Filesystem	5,207	0	0	1	1
Total indep	603,232	154	76	58	288
Arch-dep. core	29,237	4,777	0	1	4,778

Table 4.2: Number of lines modified in the Linux kernel

cases, the interface between the architecture independent and dependent code changed. For example, system call handlers rely upon the architecture-dependent code to directly modify saved program state to restart an interrupted system call. On SVA, either the system call handler code or the C library needs to restart system calls; the SVA instruction set does not provide a mechanism for the architecture dependent code to use to modify saved program state directly.

The Linux kernel needed only a small number of changes to the architecture-independent code and to device drivers for porting to SVA-OS. The drivers needed changes in order to use the instructions for I/O. The core kernel needed some changes in the signal-handler dispatch code to save state in kernel memory instead of on the user stack because the SVA-OS instructions provide no mechanism to inspect the state and ensure that the state has not been tampered. In fact, many of the changes in Table 4.2 were due to changing the name of a structure. A cleaner port may yield a kernel with even fewer changes to its architecture independent code.

4.5.2 Memory Allocator Changes

As detailed in Section 4.3.4, a kernel's memory allocators require several modifications in order to take advantage of the memory safety properties of our virtual machine.

First, we identified Linux's `kmem_cache_alloc` allocator as the only pool allocator; the rest were treated as ordinary allocators. We identified the `kmem_cache_alloc` and `kmem_cache_free` functions as allocation and deallocation routines so that the compiler instrumentation component inserts object registrations after those operations. We similarly identified the allocation/deallocation routines for the other allocators.

Second, we added a routine to support dynamic allocation throughout the kernel's lifetime (for stack objects promoted to the heap). This uses `_alloc_bootmem` early in the boot stages and then uses `kmalloc`.

Third, all kernel allocators must refrain from returning their physical page frames back to the system until the SVM indicates that it is safe to do so (the SVM will do this when the metapool is destroyed). For Linux, we modified `kmem_cache_create` to mark all pools with the `SLAB_NO_REAP` flag to prevent the buddy allocator from reclaiming unused memory from the pools when it is low on memory. We are still working on providing similar functionality for memory allocated by `vmalloc`.

Fourth, all kernel allocators must properly space objects to prevent type conversions when accessing dangling pointers. The Linux memory allocators already do this, so no changes were necessary.

Fifth, kernel allocators must zero memory before returning it. We modified `kmem_cache_alloc` to zero memory and call the pool's constructor function, if present. We are investigating what needs to be done for other kernel allocators.

Finally, as explained in Section 4.3.3, we exposed the relationship between `kmalloc` and `kmem_cache_alloc`. The former is simply implemented as a collection of caches for different sizes. By exposing this, the compiler only needs to merge metapools that correspond to each cache instead of all those corresponding to any `kmalloc`.

The number of changes required for the Linux allocators are shown in the fourth column of Table 4.2. The required changes are localized to a single file in the core kernel and are trivial to add.

4.5.3 Changes to Improve Analysis

We made several changes to the kernel source code to help improve the precision of the analysis, including the changes to eliminate unanalyzable int-to-pointer casts as explained in Section 4.3.7. First, we rewrote function signatures to reflect more accurately the type of the arguments. Second, we rewrote some structures declared as unions to use explicit structures. Last, we rewrote hard-to-analyze stack usage for accessing task structures.

Several functions, most notably the *sys_ioctl* related ones, have parameters that are interpreted as both ints and pointers, depending on the values of other arguments. In the case of *sys_ioctl*, in fact, the argument is almost always used as a pointer into userspace.We changed the parameter (and a few other similar locations) to all related functions in the kernel to be a pointer; those few locations that treat it as an integer cast from pointer to integer. This is sufficient because casts from pointer to integer do not affect the pointer analysis whereas casts from integers to pointers look like unanalyzable manufactured pointers.

Some important structures in the kernel were declared in a way that obscured the type of the structure. The most notable example is the initial task structure. This structure is declared as a union of a task structure

and an array which acts as the initial kernel stack. Changing this to a struct with the task structure and a smaller array, which reserves the same amount of memory, makes the analysis better able understand the type, increasing precision in the analysis.

The last major change was in changing how the current task structure is accessed. The kernel performed masking operations of the stack pointer to find the structure. This is very hard to analyze and was changed into an easier to analyze global variable that points to the current task structure.

4.6 Experimental Results

Our experiments aim to evaluate three aspects of SVA-M: the performance overhead of SVA-M due to the SVA-OS instructions and the run-time safety checks; the effectiveness of our approach in catching exploits in different kernel subsystems; and an understanding of what fraction of the kernel obtains the strongest (type-safe, complete) and weakest (incomplete) safety guarantees.

4.6.1 Performance Overheads

We evaluated SVA-M's performance overheads using the HBench-OS microbenchmark suite [31] and a set of standard user-level applications. We report performance metrics for four versions of the Linux 2.4.22 kernel:

1. *Linux-native*: the original kernel compiled directly to native code with GCC 3.3 [4].
2. *Linux-SVA-GCC*: the SVA-ported kernel (see Section 4.5) compiled with GCC 3.3.
3. *Linux-SVA-LLVM*: the SVA-ported kernel compiled with the LLVM C compiler.
4. *Linux-SVA-Safe*: the SVA-ported kernel compiled with the LLVM C compiler with the run-time check instrumentation pass enabled.

Because SVA-OS is implemented simply as a C library that can be linked with the SVA-ported Linux kernel, that kernel can be compiled with any C compiler. Thus, Linux-SVA-GCC and Linux-SVA-LLVM are simply that kernel compiled with the GCC and LLVM C compilers, respectively. The Linux-SVA-Safe version is nearly the same as the Linux-SVA-LLVM one with the additional run-time checks inserted. The only other difference is that Linux-SVA-Safe also has the two compiler *transformations* described in Section 4.3.8 to make alias analysis results more accurate: function cloning and function devirtualization. We expect the performance impact of these to be relatively small (although the precision improvement in pointer analysis

[4] We found that GCC4 miscompiled the native kernel.

can be significant, these pointer analysis results are only used for safety checking and are *not* used for any optimizations). To generate native code for the Linux-SVA-LLVM and Linux-SVA-Safe kernels, we translate LLVM bytecode to C code and compile the output with GCC4 -O2 (this factors out the difference between the GCC and LLVM native back-ends).

All four kernels are configured identically, in SMP mode with TCP/IPv4, Ext2, Ext3 and ISO 9660 filesystems, and a few drivers. For Linux-SVA-Safe, some portions of the kernel were not processed by the run-time check instrumentation pass because of errors encountered in booting the kernel. These were the memory subsystem (`mm/mm.o` and `arch/llvm/mm/mm.o`), two sets of utility libraries (`lib/lib.a` and `arch/llvm/lib/lib.a`), and the character drivers.

We ran all of our experiments on an 800 MHz Pentium III machine with a 256KB L2 cache and 256MB of RAM. While this is an old machine, we expect little change in relative performance overhead on newer x86 processors.

Application Performance Impact

The results for application tests are shown in Table 4.3. We used four "local" applications (top half of Table 4.3 showing the mean of 10 runs): two SPEC benchmarks, the MP3 encoder `lame`, and `ldd` (a kernel-intensive utility that prints the libraries needed by an executable). Column 2 shows that these programs spend widely varying percentages of time executing in the kernel, with `ldd` being an extreme case for local programs. We also used two servers: OpenSSH's `sshd` and `thttpd` (bottom half of Table 4.3 showing the median of 3 runs). For the local applications, we measured the elapsed time using a remote machine's clock so as to not rely on the experimental kernel's time keeping facilities. For `sshd`, we used the `scp` command from a remote machine to measure the time needed to transfer a 42 megabyte file. For `thttpd`, we used ApacheBench to measure the total time to transfer various files over 25 simultaneous connections. All server tests were executed over an isolated 100 Mb Ethernet network.

Table 4.3 shows the execution time with the Linux-native kernel and the overheads when using each of the other three kernels as a percentage of the Linux-native kernel's execution time (i.e., $100 \times (T_{other} - T_{native})/T_{native}\%$). Using the numbers for each of the latter three kernels and subtracting the preceding column (using 0% for Linux-native) isolates the impact of the SVA-OS instructions alone, of using the LLVM C compiler instead of GCC, and of introducing the safety checks.

Three of the local applications show little overhead, including `bzip2`, which spends 16% of its time in the kernel. Only `ldd` shows significant overhead, most likely due to its heavy use of `open`/`close` (see below).

Test	% System Time	Native (s)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
bzip2 (8.6MB)	16.4	11.1	1.8	0.9	1.8
lame (42MB)	0.91	12.7	0.8	0.0	1.6
gcc (-O3 58klog)	4.07	24.3	0.4	1.2	2.1
ldd (all system libs)	55.9	1.8	44.4	11.1	66.7
scp (42MB)	-	9.2	0.00	0.00	-1.09
thttpd (311B)	-	1.69	10.1	13.6	61.5
thttpd (85K)	-	36.1	0.00	-0.03	4.57
thttpd (cgi)	-	19.4	8.16	9.40	37.2

Table 4.3: Application latency increase as a percentage of Linux native performance.

`thttpd` request	# Requests	Native (KB/s)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
311 B	5k	1482	3.10	4.59	33.3
85 KB	5k	11414	0.21	-0.26	2.33
cgi	1k	28.3	-0.32	-0.46	21.8

`thttpd` Bandwidth reduction as a percentage of Linux native performance (25 concurrent connections)

Table 4.4: `thttpd` Bandwidth reduction

The network benchmarks, generally, show greater overhead. The worst case, with a 62% slowdown, is `thttpd` serving a small file to many simultaneous connections. When serving larger files, the overhead drops considerably, to 4.6%. Note that the slowdown due to the SVA-OS instructions alone (Linux-SVA-GCC vs. Linux-native) is very small for `thttpd`; most of the penalty comes from the safety checking overheads. The `sshd` server shows no overhead at all (the apparent speedup from safety checks is within the range of measurement error).

Table 4.4 shows that the total impact on bandwidth for `thttpd` is reasonable, with reductions below 34%. Most of the overhead stems from our safety checks.

Kernel Microbenchmarks

To investigate the impact of SVA-M on kernel performance in more detail, we used several latency and bandwidth benchmarks from the HBench-OS suite [31]. We configured HBench-OS to run each test 50 times and measured time using the processor's cycle counters. Tables 4.5 and 4.6 show the mean of the 50 runs.

Overall file bandwidth has small overhead (8%) while pipe bandwidth overhead is higher (67%). The latency results in Table 4.5 show many moderate overheads of 20-56%, but there are a few severe cases of overheads reaching 2x-4x.

Test	Native (μs)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
getpid	0.38	21.1	21.1	28.9
getrusage	0.63	39.7	27.0	42.9
gettimeofday	0.61	47.5	52.5	55.7
open/close	2.97	14.8	27.3	386
sbrk	0.53	20.8	26.4	26.4
sigaction	0.86	14.0	14.0	123
write	0.71	39.4	38.0	54.9
pipe	7.25	62.8	62.2	280
fork	106	24.9	23.3	74.5
fork/exec	676	17.7	20.6	54.2

Table 4.5: Latency increase for raw kernel operations as a percentage of Linux native performance

The difference between the LLVM and GCC code generators creates at most a 13% overhead. Most of the overhead comes from the use of the SVA-OS instructions and the run-time safety checks. For system calls that do little processing, the SVA-OS instructions cause the most overhead. Perhaps not surprisingly, run-time checks tend to add the most overhead to system calls that perform substantially more computation, e.g., open/close, pipe, fork and fork/exec.

Future Performance Improvements

The performance experiments above only provide a snapshot showing the current performance of the SVA-M prototype. The overall system design effort has been quite large, and therefore, we have only had time to do preliminary performance tuning of the run-time check instrumentation pass and run-time system in SVA-M. There are still at least three major improvements that we expect to make to reduce the overheads of the run-time checks:

1. using "fat pointers" instead of splay tree lookups for pointer variables in *complete* partitions, which are completely internal to the kernel being compiled (avoiding the compatibility problems fat pointers can cause when being passed to or from external code);

2. better compile-time optimizations on bounds-checks, especially hoisting checks out of loops with monotonic index ranges (a common case); and

3. performing static array bounds checking to reduce run-time checks on `getelementptr` operations.

Test	Native (MB/s)	SVA gcc (%)	SVA llvm (%)	SVA Safe (%)
file read (32k)	407	0.80	1.07	1.01
file read (64k)	410	0.69	0.99	0.80
file read (128k)	350	5.15	6.10	8.36
pipe (32k)	567	29.4	31.2	66.4
pipe (64k)	574	29.1	31.0	66.5
pipe (128k)	315	12.5	17.4	51.4

Table 4.6: Bandwidth reduction for raw kernel operations as a percentage of Linux native performance

4.6.2 Exploit Detection

To see how well our system detects exploits that use memory error vulnerabilities, we tried five different exploits on our system that were previously reported for this version of the Linux kernel, and which occur in different subsystems of the kernel. We were limited to five because we had to choose ones that were memory error exploits and for which working proof of concept code existed. Only one exploit was in a device driver; the rest were in the IPv4 network module (two exploits), the Bluetooth protocol module, and the ELF loader in the core filesystem module.

The SVA-M checks caught four out of the five exploits. Two of these were integer overflow errors in which too small a heap object was allocated, causing an array overrun [72, 137]. A third was a simple out of bounds error when indexing into a global array [146]. Finally, SVA-M caught a buffer overrun caused by decrementing a length byte to a negative value and then using that value as an unsigned array length (making it a large positive number) [136].

SVA-M did not catch the exploit in the ELF loader [135]. This one caused the kernel's user-to-kernel copying routine to overflow a kernel object by using an unchecked negative value (interpreted as a large positive value) as the object length. SVA-M failed to catch this because the implementation of the user-to-kernel copying function was in a kernel library that was not included when running the run-time check instrumentation pass on the kernel. We anticipate that including that library will allow SVA-M to catch this exploit.

4.6.3 Analysis Results

To get a sense of how many accesses receive the different levels of security, we examined the static number of instructions that access type safe metapools (the highest level) and the number that access incomplete metapools (the lowest level). These represent the two extremes of our safety guarantees. However, not all

	Allocation Sites Seen	**Access Type**	**Incom-plete**	**Type Safe**
Kernel As Tested	99.3%	Loads	80%	29%
		Stores	75%	32%
		Structure Indexing	91%	16%
		Array Indexing	71%	41%
Entire Kernel	100%	Loads	0%	26%
		Stores	0%	34%
		Structure Indexing	0%	12%
		Array Indexing	0%	39%

Table 4.7: Static metrics of the effectiveness of the safety-checking compiler

accesses are equally easy to exploit. Therefore we considered four cases: loads, stores, structure indexing ($struct.field$), and array indexing ($array[index]$). Buffer overflows fall into this last category. The first kernel presented in Table 4.7 is the kernel used in the performance and safety experiments; the second one includes the complete kernel. No sources of incompleteness remain in the second kernel because all entry points are known to the analysis, userspace is considered a valid object for syscall parameters, and all SVA-M operations are understood. Also, in both cases, no unanalyzable casts to pointers remained.

Column 4 in Table 4.7 shows that in the first kernel, 71%–91% of different kinds of accesses are to incomplete nodes, i.e., may be accessing unregistered objects. The figure is 71% for array indexing operations, the most common cause of memory errors. The second column in the table also shows, however, that over 99% of dynamic allocation sites in the kernel were instrumented; most of the rest are objects used internally by the allocators (which are within the memory subsystem). This means that almost all objects will be checked. (The fraction of accesses to incomplete *nodes* may nevertheless be high because many points-to graph nodes are likely to be exposed to the memory system and low-level libraries.) In the complete kernel, there are no unregistered objects, so all objects will be checked.

Table 4.7 also shows that much of the kernel is not type safe. This reduces the ability to eliminate load-store checks and increases the cost of the splay tree lookups because the pointer analysis may not be able to generate fine-grained metapools. The level of type safety did not vary much between the complete and incomplete kernels. We believe two types of changes can greatly improve these numbers: additional porting effort (to reduce non-type-safe C idioms) and several key refinements to the type merging properties in our pointer analysis. Again, these are both tuning exercises that have not yet been done for SVA or the ported kernel.

4.7 Related Work

Since the early work on system virtualization on the IBM S/370, there have been numerous systems (called hypervisors or Virtual Machine Monitors) that allow multiple, complete OS instances on a single hardware system. These include recent systems like VMware [147], Denali [152], Xen [60], QEMU [8], and others. The SVA-M approach is orthogonal to the hypervisor based virtualization approach. The two approaches could be combined to achieve new properties such as memory safety for the hypervisor itself.

There have been several experimental operating systems written using safe languages [28, 70, 122, 74, 53, 27, 77, 73, 156]. It is difficult to rewrite today's commodity systems in such languages. Furthermore, all these systems rely on garbage collection to manage memory. Retrofitting garbage collection into commodity systems is quite difficult since many reachable heap-allocated objects would get leaked. In contrast, SVA-M is designed to provide a safe execution environment to commodity kernels directly, requiring no language changes and preserving the kernel's explicit and low-level memory management techniques.

Numerous projects have focused on isolated execution of application-specific extensions or specific kernel components (especially device drivers) within a commodity kernel [95, 29, 148, 112, 141, 144, 164, 34]. While isolating faults is useful for reliability and restart, it does not protect operating system kernels or applications from malicious and intentional memory safety attacks. Furthermore, as Chapters 4 and 5 show, many bugs are in the core kernel code. Isolation techniques typically are not designed to protect core kernel code; instead, they typically protect the core code from extension code.

While some of the above systems for isolation use interpreters [95] or coarse-grained compartmentalization of memory [148, 141, 144], a few others enforce fine-grained safety via language or compiler techniques applicable to commodity kernels [112, 29, 164]. Proof-carrying code [112] provides an efficient and safe way to write kernel extensions and can be used with a variety of proof formalisms for encoding safety or security properties of programs [110]. The approach, however, relies on type-safety (e.g., the PCC compiler for C was limited to a type-safe subset of C with garbage collection [109]) and appears difficult to extend to the full generality of (non-type-safe) C used in commodity kernels.

The Open Kernel Environment (OKE) allows custom extensions written in the Cyclone language[80] to be run in the kernel [29]. This approach is difficult to extend to a complete commodity kernel because Cyclone has many syntactic differences from C that are required for enforcing safety, and (as the OKE authors demonstrate), Cyclone introduces some significant implementation challenges within a kernel, including the need for custom garbage collection [29].

The SafeDrive project (like Cyclone) provides fine-grained memory and type safety within system extensions, although their only reported experience is with device drivers [164]. SafeDrive's type system (called Deputy) requires annotations at external entry points (and potentially other interfaces) to identify the bounds of incoming pointer variables. In contrast to both Cyclone and SafeDrive, SVA-M provides fine-grained safety guarantees both for the core kernel as well as device drivers and avoids the need for annotations.

Some static analysis techniques, such as Engler et. al.'s work [62], have targeted bugs in OS kernels. These techniques are able to find a variety of programing mistakes, from missing null pointer checks to lock misuse. However, these techniques are not able to provide guarantees of memory safety, as SVA-M aims to do. These techniques are complementary to runtime memory safety enforcement because they can be used to *eliminate* some of the errors that SVA-M would only discover at run-time, and they can address broader classes of errors beyond memory and type safety.

TALx86 [106] is a typed assembly language (TAL) for the x86 instruction set that can express type information from rich high-level languages and can encode safety properties on native x86 code. While TALx86 has broader goals than memory safety (e.g., it can encode reliability guarantees also), it assumes garbage collection for encoding any type system with safety guarantees. SVA-M, in contrast, avoids garbage collection through a combination of run-time checks and a type-safety inference algorithm that takes into account memory reuse within kernel pools. Furthermore, any type information in TALx86 has to correctly handle many low-level features such as callee-saved registers, many details of the call stack, computed branch addresses (which require typing preconditions on branch target labels), and general-purpose registers (GPRs) that can hold multiple types of values. *None of these features arise with SVA-M* which greatly simplifies the tasks of defining and implementing the encoding of security properties. Overall, we believe SVA-M provides a more attractive foundation for encoding multiple security properties in (virtual) object code and verifying them at the end-user's system. It could be combined with a lower-level layer like TALx86 to verify these properties on the generated native code as well, taking the translator out of the trusted computing base.

The system most related to SVA-M is HyperSafe [149]. HyperSafe enforces control-flow integrity [10] on a simple hypervisor to prevent exploitation via control-flow alteration. HyperSafe also controls MMU configuration to prevent modifications to the hypervisor's code segment [149]. SVA-M provides the same safety guarantees as HyperSafe, but it also provides support for sound points-to analysis and type-inference; this, in turn, protects the software stack from a wider range of memory-safety attacks (such as dangling pointer attacks [14] and attacks that do not divert control-flow [37]). Additionally, SVA-M is designed to protect the entire software stack whereas HyperSafe only protects a single hypervisor.

4.8 Summary

Secure Virtual Architecture (SVA) defines a virtual instruction set, implemented using a compiler-based virtual machine, suitable for a commodity kernel and ordinary applications. SVA-M, an implementation of SVA, uses a novel strategy to enforce a safe execution environment for both kernel and application code. The approach provides many of the benefits of a safe language like Modula-3, Java, or C# without sacrificing the low-level control over memory layout and memory allocation/deallocation enabled by C code in commodity kernels. Furthermore, the design utilizes type-checking to remove sophisticated compiler analyses such as points-to analysis from the trusted computing base. Our experiments with 5 previously reported memory safety exploits for the Linux 2.4.22 kernel (for which exploit code is available) show that SVA-M is able to prevent 4 out of the 5 exploits and would prevent the fifth one simply by compiling an additional kernel library.

Chapter 5

Secure Low-Level Software/Hardware Interactions

5.1 Introduction

As discussed in Chapter 4, there is a growing body of work on using language and compiler techniques to enforce *memory safety* (defined in Section 5.2) for OS code. These include new OS designs based on safe languages [156, 28, 70, 78, 122], compiler techniques such as SVA-M to enforce memory safety for commodity OSs in unsafe languages, and instrumentation techniques to isolate a kernel from extensions such as device drivers [144, 148, 164]. We use the term "*safe execution environment*" (again defined in Section 5.2) to refer to the guarantees provided by a system that enforces memory safety for operating system code. Singularity [78], SPIN [28], JX [70], JavaOS [122], SafeDrive [164], and SVA-M are examples of systems that enforce a safe execution environment.

Unfortunately, all these memory safety techniques (with the exception of Verve [156], which has very limited I/O and no MMU support) make assumptions that are routinely violated by low-level interactions between an OS kernel and hardware (even if implemented in a safe programming language). Such assumptions include a static, one-to-one mapping between virtual and physical memory, an idealized processor whose state is modified only via visible program instructions, I/O operations that cannot overwrite standard memory objects except input I/O targets, and a protected stack modifiable only via load/store operations to local variables. For example, when performing type checking on a method, a safe language like Java or Modula-3 or compiler techniques like those in SVA-M assume that pointer values are only defined via visible program operations. In a kernel, however, a *buggy* kernel operation might overwrite program state while it is off-processor, and that state might later be swapped in between the definition and the use of the pointer value, a *buggy* MMU mapping might remap the underlying physical memory to a different virtual page holding data of a different type, or a *buggy* I/O operation might bring corrupt pointer values into memory.

In fact, as described in Section 5.7.1, we have injected bugs into the Linux kernel ported to SVA-M that are capable of *disabling the safety checks that prevented 3 of the 4 exploits* that were detected by SVA-M

in the experiments reported in Chapter 4: the bugs modify the metadata used to track array bounds and thus allow buffer overruns to go undetected. Similar vulnerabilities can be introduced with other bugs in low-level operations. For example, there are reported MMU bugs [23, 134, 138] in previous versions of the Linux kernel that are logical errors in the MMU configuration and could lead to kernel exploits.

A particularly nasty and relatively recent example was an insidious error in the Linux 2.6 kernel (not a device driver) that led to severe (and sometimes permanent) corruption of the e1000e network card [45]. The kernel was overwriting I/O device memory with the x86 `cmpxchg` instruction, which led to corrupting the hardware. This bug was caused by a write through a dangling pointer to I/O device memory. This bug took weeks of debugging by multiple teams to isolate. A strong memory safety system should prevent or constrain such behavior, either of which would have prevented the bug.

All these problems can, in theory, be prevented by moving some of the kernel-hardware interactions into a virtual machine (VM) and providing a high-level interface for the OS to invoke those operations safely. If an OS is *co-designed* with a virtual machine implementing the underlying language, e.g., as in JX [70], then eliminating such operations from the kernel could be feasible. For commodity operating systems such as Linux, Mac OS X, and Windows, however, reorganizing the OS in such a way may be difficult or impossible, requiring, at a minimum, substantial changes to the OS design. For example, in the case of SVA-M, moving kernel-hardware interactions into the SVA VM would require extensive changes to any commodity system ported to SVA.

Virtual machine monitors (VMMs) such as VMWare or Xen [60] do not solve this problem. They provide sufficiently strong guarantees to enforce isolation and fair resource sharing between different OS instances (i.e., different "domains") but do not enforce memory safety *within* a single instance of an OS. For example, a VMM prevents one OS instance from modifying memory mappings for a different instance but does not protect an OS instance from a bug that maps multiple pages of its own to the same physical page, thus violating necessary assumptions used to enforce memory safety. In fact, a VMM would not solve any of the reported real-world problems listed above.

In this chapter, we present a set of novel techniques to prevent low-level kernel-hardware interactions from violating memory safety in an OS executing in a safe execution environment. There are two key aspects to our approach: (1) we define carefully a set of abstractions (an API) between the kernel and the hardware that enables a lightweight run-time checker to protect hardware resources and their behaviors; and (2) we leverage the existing safety checking mechanisms of the safe execution environment to *optimize* the extra checks that are needed for this monitoring. Some examples of the key resources that are protected

by our API include processor state in CPU registers; processor state saved in memory on context-switches, interrupts, or system calls; kernel stacks; memory-mapped I/O locations; and MMU configurations. Our design also permits limited versions of self-modifying code that should suffice for most kernel uses of the feature. Most importantly, our design provides these assurances while leaving essentially all the *logical control* over hardware behavior in the hands of the kernel, i.e., no policy decisions or complex mechanisms are taken out of the kernel. Although we focus on preserving memory safety for commodity operating systems, these principles would enable any OS to reduce the likelihood and severity of failures due to bugs in low-level software-hardware interactions.

We have incorporated these techniques in the SVA-M prototype and correspondingly modified the Linux 2.4.22 kernel previously ported to SVA. Our new techniques required a significant redesign of SVA-OS. The changes to the Linux kernel were generally simple changes to use the new SVA-OS API, even though the new API provides much more powerful protection for the entire kernel. We had to change only about 100 lines in the SVA kernel to conform to the new SVA-OS API.

We have evaluated the ability of our system to prevent kernel bugs due to kernel-hardware interactions, both with real reported bugs and injected bugs. Our system prevents two MMU bugs in Linux 2.4.22 for which exploit code is available. Both bugs crash the kernel when run under the original SVA-M. Moreover, as explained in Section 5.7.1, we would also prevent the e1000e bug in Linux 2.6 if that kernel is run on our system. Finally, the system successfully prevents all the low-level kernel-hardware interaction errors we have tried to inject.

We also evaluated the performance overheads for two servers and three desktop applications (two of which perform substantial I/O). Compared with the original SVA-M, the new techniques in this chapter add very low or negligible overheads. Combined with the ability to prevent real-world exploits that would be missed otherwise, it clearly seems worthwhile to add these techniques to an existing memory safety system.

To summarize, the key contributions of this chapter are:

- We have presented novel mechanisms to ensure that low-level kernel-hardware interactions (e.g., context switching, thread creation, MMU changes, and I/O operations) do not violate assumptions used to enforce a safe execution environment.

- We have prototyped these techniques and shown that they can be used to enforce the assumptions made by a memory safety checker for a commodity kernel such as Linux. To our knowledge, no previous safety enforcement technique provides such guarantees to commodity system software.

- We have evaluated this system experimentally and shown that it is effective at preventing exploits in the above operations in Linux while incurring little overhead over and above the overhead of the underlying safe execution environment of SVA-M.

5.2 Breaking Memory Safety with Low-Level Kernel Operations

Informally, *a program is type-safe* if all operations in the program respect the types of their operands. For the purposes of this work, we say *a program is memory safe* if every memory access uses a previously initialized pointer variable, accesses the same object to which the pointer pointed initially,[1] and the object has not been deallocated. Memory safety is necessary for type safety (conversely, type safety implies memory safety) because dereferencing an uninitialized pointer, accessing the target object out of bounds, or dereferencing a dangling pointer to a freed object can all cause accesses to unpredictable values and hence allow illegal operations on those values.

A safe programming language guarantees type safety and memory safety for all legal programs [123]; these guarantees also imply a *sound operational semantics* for programs in the language. Language implementations enforce these guarantees through a combination of compile-time type checking, automatic memory management (e.g., garbage collection or region-based memory management) to prevent dangling pointer references, and run-time checks such as array bounds checks and null pointer checks.

Four recent compiler-based systems for C, namely, CCured [111], SafeDrive [164], SAFECode [57], and SVA-M, enforce similar but weaker guarantees for C code. Their guarantees are weaker in two ways: (a) they provide type safety for only a subset of objects, and (b) three of the four systems — SafeDrive, SAFECode, and SVA-M — permit dangling pointer references (use-after-free) to avoid the need for garbage collection. Unlike SafeDrive, however, SAFECode and SVA-M *guarantee* that dangling pointer references do not invalidate any of the other safety properties, i.e., partial type safety, memory safety, or a sound operational semantics [56, 57]. We refer to all these systems – safe languages or safety checking compilers – as providing a *safe execution environment*.

All of the above systems make some fundamental assumptions regarding the run-time environment in enforcing their safety guarantees. In particular, these systems assume that the code segment is static, control flow can only be altered through explicit branch instructions, call instructions, and visible signal handling, and that data is stored either in a flat, unchanging address space or in processor registers. Furthermore, data can only be read and written by direct loads and stores to memory or direct changes to processor registers.

[1] We permit a pointer to "leave" its target object and later return as long as it is not accessed while it is out of bounds [120].

Low-level system code routinely violates these assumptions. Operating system kernels, virtual machine monitors, language virtual machines such as a JVM or CLR, and user-level thread libraries often perform operations such as context switching, direct stack manipulation, memory mapped I/O, and MMU configuration, that violate these assumptions. More importantly, as explained in the rest of this section, perfectly *type-safe* code can violate many of these assumptions (through logical errors), i.e., such errors will not be prevented by the language in the first place. This is unacceptable for safe language implementations and, at least, undesirable for system software because these violations can compromise safety and soundness and thus permit the vulnerabilities a safe language was designed to prevent, such as buffer overflows or the creation of illegal pointer values.

There are, in fact, a small number of root causes (or categories of root causes) of all these violations. This section enumerates these root causes, and the next section describes the design principles by which these root causes can be eliminated. We assume throughout this discussion that a safety checker (through some combination of static and run-time checking) enforces the language-level safety guarantees of a safe execution environment, described above, for the kernel.[2] This allows us to assume that the run-time checker itself is secure and that static analysis can be used soundly on kernel code [57]. Our goal is to ensure the integrity of the *assumptions* made by this safety checker. We refer to the extensions that enforce these assumptions as a *verifier*.

Briefly, the fundamental categories of violations are:

- corrupting processor state when held in registers or memory;
- corrupting stack values for kernel threads;
- corrupting memory mapped I/O locations;
- corrupting code pages in memory;
- other violations that can corrupt arbitrary memory locations, including those listed above.

Unlike the last category, the first four above are errors that are specific to individual categories of memory.

5.2.1 Corrupting Processor State

Corrupting processor state can corrupt both data and control flow. The verifier must first ensure that processor state cannot be corrupted while on the processor itself, i.e., preventing arbitrary changes to processor

[2] This chapter focuses on enforcing memory safety for the kernel. The same techniques could be applied to protect user-space threads from these violations.

registers. In addition, however, standard kernels save processor state (i.e., data and control registers) in memory where it is accessible by standard (even type-safe) load and store instructions. Any (buggy) code that modifies this state before restoring the state to the processor can alter control flow (the program counter, stack pointer, return address register, or condition code registers) or data values. In safe systems that permit dangling pointer references, processor state can also be corrupted if the memory used to hold saved processor state (usually located on the heap [30]) is freed and reallocated for other purposes.

Note that there are cases where the kernel makes explicit, legal, changes to the interrupted state of user-space code. For example, during signal handler dispatch, the kernel modifies interrupted program state that has been saved to memory, including the interrupted program's program counter and stack pointer [30]. Also, returning from a signal handler requires undoing the modifications made by signal delivery. The verifier must be able to distinguish legal from illegal changes to saved state.

5.2.2 Corrupting Stack State

The kernel directly manages the stacks of both user and kernel threads; it allocates and deallocates memory to hold them, sets up initial stack frames for new threads and signal handlers, and switches between stacks during a context switch or interrupt/system call return.

Memory for the stack is obtained from some standard memory allocation. Several safety violations are possible through this allocated memory. First, the memory for the stack should only be used for stack frames created during normal function calls and not directly modified via arbitrary stores;[3] such stores could corrupt the stack frames and thus compromise safety. Second, the memory for the stack must not be deallocated and reused for other memory objects while the stack is still in use. Third, a context switch must switch to a stack and its corresponding saved processor state as a pair; a context switch should not load processor state with the wrong stack or with a stack that has been deallocated. Fourth, after a stack is deallocated, live pointers to local variables allocated on the stack must not be dereferenced (the exiting thread may have stored pointers to such objects into global variables or the heap where they are accessible by other threads).

5.2.3 Corrupting Memory-Mapped I/O

Most systems today use memory-mapped I/O for controlling I/O devices and either memory-mapped I/O or DMA for performing data transfers. Many hardware architectures treat regular memory and memory-mapped I/O device memory (hereafter called I/O memory) identically, allowing a single set of hardware

[3] An exception is when Linux stores the process's task structure at the bottom of the stack [30].

instructions to access both. From a memory safety perspective, however, it is better to treat regular memory and I/O memory as disjoint types of memory that are accessed using distinct instructions. First, I/O memory is not semantically the same as regular memory in that a load may not return the value last stored into the location; program analysis algorithms (used to enforce and optimize memory safety [57]) are not sound when applied to such memory. Second, I/O memory creates side-effects that regular memory does not. While erroneously accessing I/O memory instead of regular memory may not be a memory safety violation per se, it is still an error with potentially dire consequences. For example, the e1000e bug [45] caused fatal damage to hardware when an instruction (`cmpxchg`) that was meant to write to memory erroneously accessed memory-mapped I/O registers, which has undefined behavior. Therefore, for soundness of regular memory safety and for protection against a serious class of programming errors, it is best to treat regular memory and I/O memory as disjoint.

5.2.4 Corrupting Code

Besides the general memory corruption violations described below, there are only two ways in which the contents of code pages can be (or appear to be) corrupted. One is through self-modifying code (SMC); the other is through incorrect program loading operations (for new code or loadable kernel modules).

Self-modifying code directly modifies the sequence of instructions executed by the program. This can modify program behavior in ways not predicted by the compiler and hence bypass any of its safety checking techniques. For these reasons, most type-safe languages prohibit self-modifying code (which is distinct from "self-extending" behaviors like dynamic class loading). However, modern kernels use limited forms of self-modifying code for operations like enabling and disabling instrumentation [45] or optimizing synchronization for a specific machine configuration [44]. To allow such optimizations, the verifier must define limited forms of self-modifying code that do not violate the assumptions of the safety checker.

Second, the verifier must ensure that any program loading operation is implemented correctly. For example, any such operation, including new code, self-modifying code, or self-extending code (e.g., loadable kernel modules) requires flushing the instruction cache. Otherwise, cached copies of the old instructions may be executed out of the I-cache (and processors with split instruction/data caches may even execute old instructions with fresh data). This may lead to arbitrary memory safety violations for the kernel or application code.

5.2.5 General Memory Corruption

Finally, there are three kinds of kernel functionality that can corrupt arbitrary memory pages: (1) MMU configuration; (2) page swapping; and (3) DMA. Note that errors in any of these actions are generally invisible to a safety checking compiler and can violate the assumptions made by the compiler, as follows.

First, the kernel can violate memory safety with direct operations on virtual memory. Fundamentally, most of these are caused by creating an incorrect virtual-to-physical page mapping. Such errors include modifying mappings in the range of kernel stack memory, mapping the same physical page into two virtual pages (unintentionally), and changing a virtual-to-physical mapping for a live virtual page. As before, any of these errors can occur even with a type-safe language.

A second source of errors is in page swapping. When a page of data is swapped in on a page fault, memory safety can be violated if the data swapped in is not identical to the data swapped out from that virtual page. For example, swapping in the wrong data can cause invalid data to appear in pointers that are stored in memory.

Finally, a third source of problems is DMA. DMA introduces two problems. First, a DMA configuration error, device driver error, or device firmware error can cause a DMA transfer to overwrite arbitrary physical memory, violating type-safety assumptions. Second, even a correct DMA transfer may bring in unknown data which cannot be used in a type-safe manner, unless special language support is added to enable that, e.g., to prevent such data being used as pointer values, as in the SPIN system [76].

5.3 Design Principles

We now describe the general design principles that a memory safe system can use to prevent the memory errors described in Section 5.2. As described earlier, we assume a safety checker already exists that creates a safe execution environment; the *verifier* is the set of extensions to the safety checker that enforces the underlying assumptions of the checker. Examples of safety checkers that could benefit directly from such extensions include SVA-M, SafeDrive [164], and XFI [144]. We also assume that the kernel source code is available for modification.

Processor State: Preventing the corruption of processor state involves solving several issues. First, the verifier must ensure that the kernel does not make arbitrary changes to CPU registers. Most memory safe systems already do this by not providing instructions for such low-level modifications. Second, the verifier must ensure that processor state saved by a context switch, interrupt, trap, or system call is not accessed by

memory load and store instructions. To do this, the verifier can allocate the memory used to store processor state within its own memory and allow the kernel to manipulate that state via special instructions that take an opaque handle (e.g., a unique integer) to identify which saved state buffer to use. For checkers like SVA-M and SafeDrive [164], the safety checker itself prevents the kernel from manufacturing and using pointers to these saved state buffers (e.g., via checks on accesses that use pointers cast from integers). Additionally, the verifier should ensure that the interface for context switching leaves the system in a known state, meaning that a context switch should either succeed completely or fail.

There are operations in which interrupted program state needs to be modified by the kernel (e.g., signal handler dispatch). The verifier must provide instructions for doing controlled modifications of interrupted program state; for example, it can provide an instruction to push function call frames on to an interrupted program's stack as described in Chapter 2. Such instructions must ensure that either their modifications cannot break memory safety or that they only modify the saved state of interrupted user-space programs (modifying user-space state cannot violate the kernel's memory safety).

Stack State: The memory for a kernel stack and for the processor state object (the in-memory representation of processor state) must be created in a single operation (instead of by separate operations), and the verifier should ensure that the kernel stack and processor state object are always used and deallocated together. To ease implementation, it may be desirable to move some low-level, error-prone stack and processor state object initialization code into the verifier. The verifier must also ensure that memory loads and stores do not modify the kernel stack (aside from accessing local variables) and that local variables stored on the stack can no longer be accessed when the kernel stack is destroyed.

Memory-mapped I/O: The verifier must require that all I/O object allocations be identifiable in the kernel code (e.g., declared via a pseudo-allocator). It should also ensure that only special I/O read and write instructions can access I/O memory (these special instructions can still be translated into regular memory loads and stores for memory-mapped I/O machines) and that these special instructions cannot read or write regular memory objects. If the verifier uses type-safety analysis to optimize run-time checks, it should consider I/O objects (objects analogous to memory objects but that reside in memory-mapped I/O pages) to be *type-unsafe* as the device's firmware may use the I/O memory in a type-unsafe fashion. Since it is possible for a pointer to point to both I/O objects and memory objects, the verifier should place run-time checks on such pointers to ensure that they are accessing the correct type of object (memory or I/O), depending upon the operation in which the pointer is used.

Kernel Code: The verifier must not permit the kernel to modify its code segment. However, it can support a limited version of self-modifying code that is easy to implement and able to support the uses of self-modifying code found in commodity kernels. In our design, the kernel can specify regions of code that can be enabled and disabled. The verifier will be responsible for replacing native code with no-op instructions when the kernel requests that code be disabled and replacing the no-ops with the original code when the kernel requests the code to be re-enabled. When analyzing code that can be enabled and disabled, the verifier can use conservative analysis techniques to generate results that are correct regardless of whether the code is enabled or disabled. For example, our pointer analysis algorithm, like most other inter-procedural ones used in production compilers, computes a *may-points-to* result [88], which can be computed with the code enabled; it will still be correct, though perhaps conservative, if the code is disabled.

To ensure that the instruction cache is properly flushed, our design calls for the safety checker to handle all translation to native code. The safety checker already does this in JVMs, safe programming languages, and in the SVA-M system. By performing all translation to native code, the verifier can ensure that all appropriate CPU caches are flushed when new code is loaded into the system.

General Memory Corruption: The verifier must implement several types of protection to handle the general memory corruption errors in Section 5.2.5.

MMU configuration: To prevent MMU misconfiguration errors, the verifier must be able to control access to hardware page tables or processor TLBs and vet changes to the MMU configuration before they are applied. Implementations can use para-virtualization techniques [60] to control the MMU. The verifier must prevent pages containing kernel memory objects from being made accessible to non-privileged code and ensure that pages containing kernel stack frames are not mapped to multiple virtual addresses (i.e., double mapped) or unmapped before the kernel stack is destroyed.[4] Verifiers optimizing memory access checks must also prohibit double mappings of pages containing *type known* objects; this will prevent data from being written into the page in a way that is not detected by compiler analysis techniques. Pages containing type-unknown memory objects can be mapped multiple times since run-time checks already ensure that the data within them does not violate any memory safety properties. The verifier must also ensure that MMU mappings do not violate any other analysis results upon which optimizations depend.

Page swapping: For page swapping, the kernel must notify the verifier before swapping a page out (if not, the verifier will detect the omission on a subsequent physical page remapping operation). The verifier can then record any metadata for the page and compute a checksum of the page's contents; the verifier can

[4] We assume the kernel does not swap stack pages to disk, but the design can be extended easily to allow this.

then use these to verify that the page contents have not changed when the OS kernel swaps the page back into physical memory.

DMA: The verifier should prevent DMA transfers from overwriting critical memory such as the kernel's code segment, the verifier's code and data, kernel stacks (aside from local variables), and processor state objects. Implementation will require the use of IOMMU techniques like those in previous work [66, 126]. Additionally, if the verifier uses type information to optimize memory safety checks, it must consider the memory accessible via DMA as type-unsafe. This solution is strictly stronger than previous work (like that in SPIN [76]): it allows pointer values in input data whereas they do not (and they do not guarantee type safety for other input data).

Entry Points: To ensure control-flow integrity, the kernel should not be entered in the middle of a function. Therefore, the verifier must ensure that all interrupt, trap, and system call handlers registered by the kernel are the initial address of a valid function capable of servicing the interrupt, trap, or system call, respectively.

5.4 Background: Secure Virtual Architecture

The Secure Virtual Architecture (SVA) is described in Chapters 2 and 3. As those chapters describe, SVA-OS, a subset of the SVA instruction set, provides instructions designed to support an operating system's special interaction with the hardware. These include instructions for loading from/storing to I/O memory, configuring the MMU, and manipulating program state. An important property is that a kernel ported to SVA using the SVA-OS instructions *contains no assembly code*; this simplifies SVA-M's task of safety checking. Nevertheless, these instructions provide low-level hardware interactions that can generate all the problems described in Section 5.2 if used incorrectly; it is very difficult for the compiler to check their correct use in the original design. In particular, the VM does not perform any special checks for processor state objects, direct stack manipulation, memory mapped I/O locations, MMU configuration changes, or DMA operations. Also, it disallows self-modifying code.

Since the original SVA-M system does not follow the principles of Section 5.3, it's memory safety protection is incomplete. For example, we tested two [134, 138] of the three reported low-level MMU configuration errors we found for Linux 2.4.22, the kernel version ported to SVA-M (we could not try the third [23] for reasons explained in Section 5.7.1). Although both are memory safety violations, *neither of them was detected or prevented by the original SVA-M system.*

5.5 Design

Our design is an extension of SVA-M (described in Chapter 4). SVA-M provides strong memory safety guarantees for kernel code and an abstraction of the hardware that is both low-level (e.g., context switching, I/O, and MMU configuration policies are still implemented in the kernel), yet easy to analyze (because the SVA-OS instructions for interacting with hardware are slightly higher level than typical processor instructions). Below, we describe our extensions to provide memory safety in the face of errors in kernel-hardware interactions.

5.5.1 Context Switching

Previously, the SVA-M system performed context switching using the `sva.load.integer` and `sva.save.integer` instructions described in Chapter 4, which saved from and loaded into the processor the processor state (named Integer State). These instructions stored processor state in a kernel allocated memory buffer which could be later modified by memory-safe store instructions or freed by the kernel deallocator. Our new design calls for a single instruction named `sva_swap_integer` (see Table 5.1) that saves the old processor state and loads the new state in a single operation.

This design has all of the necessary features to preserve memory safety when context switching. The `sva_swap_integer` instruction allocates the memory buffer to hold processor state within the VM's memory and returns an opaque integer identifier which can be used to re-load the state in a subsequent call to `sva_swap_integer`. Combined with SVA-M's original protections against manufactured pointers, this prevents the kernel from modifying or deallocating the saved processor state buffer. The design also ensures correct deallocation of the memory buffer used to hold processor state. The VM tracks which identifiers are mapped to allocated state buffers created by `sva_swap_integer`; these memory buffer/identifier pairs are kept alive until the state is placed back on the processor by another call to `sva_swap_integer`. Once state is placed back on the processor, the memory buffer is deallocated, and the identifier invalidated to prevent the kernel from trying to restore state from a deallocated state buffer.

Finally, `sva_swap_integer` will either succeed to context switch and return an identifier for the saved processor state, or it will fail, save no processor state, and continue execution of the currently running thread. This ensures that the kernel stack and the saved processor state are always synchronized.

Name	Description
sva_swap_integer	Saves the current processor state into an internal memory buffer, loads previously saved state referenced by its ID, and returns the ID of the new saved state.
sva_declare_stack	Declares that a memory object is to be used as a new stack.
sva_release_stack	Declares that a memory object is no longer used as a stack.
sva_init_stack	Initializes a new stack.

Table 5.1: SVA Instructions for Context Switching and Thread Creation.

5.5.2 Thread Management

A thread of execution consists of a stack and a saved processor state that can be used to either initiate or continue execution of the thread. Thread creation is therefore comprised of three operations: allocating memory for the new thread's stack, initializing the new stack, and creating an initial state that can be loaded on to the processor using `sva_swap_integer`.

The VM needs to know where kernel stacks are located in order to prevent them from being written by load and store instructions. We introduce a new SVA instruction, `sva_declare_stack`, which a kernel uses to declare that a memory object will be used as a stack. During pointer analysis, any pointers passed to `sva_declare_stack` and pointers that alias with such pointers are marked with a special *DeclaredStack* flag; this flag indicates that run-time checks are needed on stores via such pointers to ensure that they are not writing into a kernel stack. The compiler, on seeing an `sva_declare_stack` instruction, will also verify, statically (via pointer analysis) if possible but at run-time if necessary, that the memory object used for the new stack is either a global or heap object; this will prevent stacks from being embedded within other stacks. After this check is done, `sva_declare_stack` will unregister the memory object from the set of valid memory objects that can be accessed via loads and stores and record the stack's size and location within the VM's internal data structures as a valid kernel stack.

To initialize a stack and the initial processor state that will use the memory as a stack, we introduce `sva_init_stack`; this instruction will initialize the stack and create a new saved Integer State which can be used in `sva_swap_integer` to start executing the new thread. The `sva_init_stack` instruction verifies (either statically or at run-time) that its argument has previously been declared as a stack using `sva_declare_stack`. When the new thread wakes up, it will find itself running within the function specified by the call to `sva_init_stack`; when this function returns, it will return to user-space at the same location as the original thread entered.

Deleting a thread is composed of two operations. First, the memory object containing the stack must be deallocated. Second, any Integer State associated with the stack that was saved on a context switch must be

invalidated. When the kernel wishes to destroy a thread, it must call the `sva_release_stack` instruction; this will mark the stack memory as a regular memory object so that it can be freed and invalidates any saved Integer State associated with the stack.

When a kernel stack is deallocated, there may be pointers in global or heap objects that point to memory (i.e., local variables) allocated on that stack. SVA-M must ensure that dereferencing such pointers does not violate memory safety. Type-unsafe stack allocated objects are subject to load/store checks and are registered with the SVA-M virtual machine as described in Chapter 4. In order for the `sva_release_stack` instruction to invalidate such objects when stack memory is reclaimed, the VM records information on stack object allocations and associates this information with the metadata about the stack in which the object is allocated. In this way, when a stack is deallocated, any live objects still registered with the virtual machine are automatically invalidated as well; run-time checks will no longer consider these stack allocated objects to be valid objects. Type-known stack allocated objects can never be pointed to by global or heap objects; SVA-M already transforms such stack allocations into heap allocations (see Chapter 4) to make dangling pointer dereferencing to type-known stack allocated objects safe [57].

5.5.3 Memory Mapped I/O

To ensure safe use of I/O memory, our system must be able to identify where I/O memory is located and when the kernel is legitimately accessing it.

Identifying the location of I/O memory is straightforward. In most systems, I/O memory is located at (or mapped into) known, constant locations within the system's address space, similar to global variables. In some systems, a memory-allocator-like function may remap physical page frames corresponding to I/O memory to a virtual memory address [30]. The insight is that I/O memory is grouped into objects just like regular memory; in some systems, such I/O objects are even allocated and freed like heap objects (e.g., Linux's `ioremap()` function [30]). To let the VM know where I/O memory is located, we must modify the kernel to use a pseudo-allocator that informs the VM of global I/O objects; we can also modify the VM to recognize I/O "allocators" like `ioremap()` just like it recognizes heap allocators like Linux's `kmalloc()` [30].

Given this information, the VM needs to determine which pointers may point to I/O memory. To do so, we modified the SVA-M points-to analysis algorithm [88] to mark the target (i.e., the "points-to set") of a pointer holding the return address of the I/O allocator with a special *I/O flag*. This also flags other pointers aliased to such a pointer because any two aliased pointers point to a common target [88].

We also modified the points-to analysis to mark I/O memory as *type-unknown*. Even if the kernel accesses I/O memory in a type-consistent fashion, the firmware on the I/O device may not. *Type-unknown* memory incurs additional run-time checks but allows kernel code to safely use pointer values in such memory as pointers.

We also extended SVA-M to record the size and virtual address location of every I/O object allocation and deallocation by instrumenting every call to the I/O allocator and deallocator functions. At run-time, the VM records these I/O objects in a per-metapool data structure that is disjoint from the structure used to record the bounds of regular memory objects. The VM also uses new run-time checks for checking I/O load and store instructions. Since I/O pointers can be indexed like memory pointers (an I/O device may have an array of control registers), the bounds checking code must check both regular memory objects and I/O memory objects. Load and store checks on regular memory pointers *without the I/O flag* remain unchanged; they only consider memory objects. New run-time checks are needed on both memory and I/O loads and stores for pointers that have both the I/O flag and one or more of the memory flags (heap, stack, global) to ensure that they only access regular or I/O memory objects, respectively.

5.5.4 Safe DMA

We assume the use of an IOMMU for preventing DMA operations from overflowing object bounds or writing to the wrong memory address altogether [12]. The SVA-M virtual machine simply has to ensure that the I/O MMU is configured so that DMA operations cannot write to the virtual machine's internal memory, kernel code pages, pages which contain type-safe objects, and stack objects.

We mark all memory objects that may be used for DMA operations as type-unsafe, similar to I/O memory that is accessed directly. We assume that any pointer that is *stored into* I/O memory is a potential memory buffer for DMA operations. We require alias analysis to identify such stores; it simply has to check that the target address is in I/O memory and the store value is of pointer type. We then mark the points-to set of the store value pointer as *type-unknown*.

5.5.5 Virtual Memory

Our system must control the MMU and vet changes to its configuration to prevent safety violations and preserve compiler-inferred analysis results. Below, we describe the mechanism by which our system monitors and controls MMU configuration and then discuss how we use this mechanism to enforce several safety properties.

Name	Description
sva_end_mem_init	End of the virtual memory boot initialization. Flags all page table pages, and mark them read-only.
sva_declare_l1_page	Zeroes the page and flags it read-only and L1.
sva_declare_l2_page	Zeroes the page and flags it read-only and L2.
sva_declare_l3_page	Puts the default mappings in the page and flags it read-only and L3.
sva_remove_l1_page	Removes the L1 flag and makes the page writeable.
sva_remove_l2_page	Removes the L2 flag and makes the page writeable.
sva_remove_l3_page	Removes the L3 flag and makes the page writeable.
sva_update_l1_mapping	Updates the mapping if the mapping belongs to an L1 page and the page is not already mapped for a type known pool, sva page, code page, or stack page.
sva_update_l2_mapping	Updates the mapping if the mapping belongs to an L2 page and the new mapping is for an L1 page.
sva_update_l3_mapping	Updates the mapping if the mapping belongs to an L3 page and the new mapping is for an L2 page.
sva_load_pagetable	Check that the physical page is an L3 page and loads it in the page table register.

Table 5.2: MMU Interface for a Hardware TLB Processor.

Controlling MMU Configuration

SVA currently provides a hardware TLB interface (as described in Chapter 2). This interface (given in Table 5.2) is similar to those used in VMMs like Xen [60] and is based off the `paravirtops` interface [155] found in Linux 2.6. The page table is a 3-level page table, and there are instructions for changing mappings at each level. In this design, the OS first tells the VM which memory pages will be used for the page table (it must specify at what level the page will appear in the table); the VM then takes control of these pages by zeroing them (to prevent stale mappings from being used) and marking them read-only to prevent the OS from accessing them directly. The OS must then use special SVA-OS instructions to update the translations stored in these page table pages; these instructions allow SVA-M to first inspect and modify translations before accepting them and placing them into the page table. The `sva_load_pagetable` instruction selects which page table is in active use and ensures that only page tables controlled by SVA-M are ever used by the processor. This interface, combined with SVA-M's control-flow integrity guarantees (see Chapter 4), ensures that SVA-M maintains control of all page mappings on the system.

Memory Safe MMU Configuration

For preventing memory safety violations involving the MMU, the VM needs to track two pieces of information. First, the VM must know the purpose of various ranges of the virtual address space; the kernel must provide the virtual address ranges of user-space memory, kernel data memory, and I/O object memory. This

information will be used to prevent physical pages from being mapped into the wrong virtual addresses (e.g., a memory mapped I/O device being mapped into a virtual address used by a kernel memory object). A special instruction permits the kernel to communicate this information to the VM.

Second, the VM must know how physical pages are used, how many times they are mapped into the virtual address space, and whether any MMU mapping makes them accessible to unprivileged (i.e., user-space) code. To track this information, the VM associates with each physical page a set of flags and counters. The first set of flags are mutually exclusive and indicate the purpose of the page; a page can be marked as: `L1` (Level-1 page table page), `L2` (Level-2 page table page), `L3` (Level-3 page table page), `RW` (a standard kernel page holding memory objects), `IO` (a memory mapped I/O page), `stack` (kernel stack), `code` (kernel or SVA code), or `svamem` (SVA data memory). A second flag, the `TK` flag, specifies whether a physical page contains *type-known* data. The VM also keeps a count of the number of virtual pages mapped to the physical page and a count of the number of mappings that make the page accessible to user-space code.

The flags are checked and updated by the VM whenever the kernel requests a change to the page tables or performs relevant memory or I/O object allocation. Calls to the memory allocator are instrumented to set the RW and, if appropriate, the TK flag on pages backing the newly allocated memory object. On system boot, the VM sets the `IO` flag on physical pages known to be memory-mapped I/O locations. The `stack` flag is set and cleared by `sva_declare_stack` and `sva_release_stack`, respectively. Changes to the page table via the instructions in Table 5.2 update the counters and the `L1`, `L2`, and `L3` flags.

The VM uses all of the above information to detect, at run-time, violations of the safety requirements in Section 5.3. Before inserting a new page mapping, the VM can detect whether the new mapping will create multiple mappings to physical memory containing *type-known* objects, map a page into the virtual address space of the VM or kernel code segment, unmap or double map a page that is part of a kernel stack, make a physical page containing kernel memory accessible to user-space code, or map memory-mapped I/O pages into a kernel memory object (or vice-versa). Note that SVA-M currently trusts the kernel memory allocators to (i) return different virtual addresses for every allocation, and (ii) not to move virtual pages from one metapool to another until the original metapool is destroyed.

5.5.6 Self-modifying Code

The new SVA-M system supports the restricted version of self-modifying code described in Section 5.3: OS kernels can disable and re-enable pre-declared pieces of code. SVA-M will use compile-time analysis carefully to ensure that replacing the code with no-op instructions will not invalidate the analysis results.

We define four new instructions to support self-modifying code. The first two instructions, `sva_begin_alt` and `sva_end_alt`, enclose the code regions that may be modified at runtime. They must be properly nested and must be given a unique identifier. The instructions are not emitted in the native code. The two other instructions, `sva_disable_code` and `sva_enable_code`, execute at runtime. They take the identifier given to the `sva_begin_alt` and `sva_end_alt` instructions. `sva_disable_code` saves the previous code and inserts no-ops in the code, and `sva_enable_code` restores the previous code.

With this approach, SVA-M can support most uses of self-modifying code in operating systems. For instance, it can support the `alternatives`[5] framework in Linux 2.6 [44] and Linux's `ftrace` tracing support [45] which disables calls to logging functions at run-time.

5.5.7 Interrupted State

On an interrupt, trap, or system call, the original SVA-M system saves processor state within the VM's internal memory and permits the kernel to use specialized instructions to modify the state via an opaque handle called the interrupt context (see Chapters 2 and 4). These instructions, which are slightly higher-level than assembly code, are used by the kernel to implement operations like signal handler dispatch and starting execution of user programs. Since systems such as Linux can be interrupted while running kernel code [30], these instructions can violate the kernel's memory safety if used incorrectly on interrupted kernel state. To address these issues, we introduce several changes to the original SVA-M design.

First, we noticed that all of the instructions that manipulate interrupted program state are either memory safe (e.g., the instruction that unwinds stack frames for kernel exception handling described in Chapter 2) or only need to modify the interrupted state of user-space programs. Hence, all instructions that are not intrinsically memory safe will verify that they are modifying interrupted user-space program state. Second, the opaque handle to the interrupt context will be made implicit so that no run-time checks are needed to validate it when it is used. We have observed that the Linux kernel only operates upon the most recently created interrupt context; we do not see a need for other operating systems of similar design to do so, either. Without an explicit handle to the interrupt context's location in memory, no validation code is needed, and the kernel cannot create a pointer to the saved program state (except for explicit integer to pointer casts, uses of which will be caught by SVA-M's existing checks as described in Chapter 4).

[5] Linux 2.6, file include/asm-x86/alternative.h

5.5.8 Miscellaneous

As described in Chapter 2 (as well as in Monroe's thesis [104]), the VM assumes control of the hardware interrupt descriptor table to ensure control-flow integrity requirements; the OS kernel must use special instructions to associate a function with a particular interrupt, trap, or system call. Similar to indirect function call checks, SVA-M can use static analysis and run-time checks to ensure that only valid functions are registered as interrupt, trap, or system call handlers.

SVA provides two sets of atomic memory instructions: `sva_fetch_and_phi` where phi is one of several integer operations (e.g., add), and `sva_compare_and_swap` which performs an atomic compare and swap. The static and run-time checks that protect regular memory loads and stores also protect these operations.

5.6 Modifications to the Linux Kernel

We implemented our design by improving and extending the original SVA-M prototype and the SVA port of the Linux 2.4.22 kernel described in Chapter 4. The previous section described how we modified the SVA-OS instructions. Below, we describe how we modified the Linux kernel to use these new instructions accordingly. We modified less than 100 lines from the original SVA kernel to port our kernel to the new SVA-OS API; the original port of the i386 Linux kernel to SVA modified 300 lines of architecture-independent code and 4,800 lines of architecture-dependent code (see Chapter 4).

5.6.1 Changes to Baseline SVA-M

The baseline SVA-M system in our evaluation (Section 5.7) is an improved version of the original SVA-M system described in Chapter 4 that is suitable for determining the extra overhead incurred by the run-time checks necessitated by the design in Section 5.5. First, we fixed several bugs in the optimization of run-time checks. Second, while the original SVA-M system does not analyze and protect the whole kernel, there is no fundamental reason why it cannot. Therefore, we chose to disable optimizations which apply only to incomplete kernel code for the experiments in Section 5.7. Third, the new baseline SVA-M system recognizes `ioremap()` as an allocator function even though it does not add run-time checks for I/O loads and stores. Fourth, we replaced most uses of the `__get_free_pages()` page allocator with `kmalloc()` in code which uses the page allocator like a standard memory allocator; this ensures that most kernel allocations are performed in kernel pools (i.e., `kmem_cache_ts`) which fulfill the requirements for allocators as described in Chapter 4.

We also modified the SVA Linux kernel to use the new SVA-OS instruction set as described below. This ensured that the only difference between our baseline SVA-M system and our SVA-M system with the low-level safety protections was the addition of the run-time checks necessary to ensure safety for context switching, thread management, MMU, and I/O memory safety.

5.6.2 Context Switching/Thread Creation

The modifications needed for context switching were straightforward. We simply modified the `switch_to` macro in Linux [30] to use the `sva_swap_integer` instruction to perform context switching.

Some minor kernel modifications were needed to use the new thread creation instructions. The original i386 Linux kernel allocates a single memory object which holds both a thread's task structure and the kernel stack for the thread [30], but this cannot be done on our system because `sva_declare_stack` requires that a stack consumes an entire memory object. For our prototype, we simply modified the Linux kernel to perform separate allocations for the kernel stack and the task structure.

5.6.3 I/O

As noted earlier, our implementation enhances the pointer analysis algorithm in SVA-M (DSA [88]) to mark pointers that may point to I/O objects. It does this by finding calls to the Linux `__ioremap()` function. To make implementation easier, we modified `ioremap()` and `ioremap_nocache()` in the Linux source to be macros that call `__ioremap()`.

Our test system's devices do not use global I/O memory objects, so we did not implement a pseudo allocator for identifying them. Also, we did not modify DSA to mark memory stored into I/O device memory as type-unknown. The difficulty is that Linux casts pointers into integers before writing them into I/O device memory. The DSA implementation does not have solid support for tracking pointers through integers i.e., it does not consider the case where an integer may, in fact, be pointing to a memory object. Implementing these changes to provide DMA protection is left as future work.

5.6.4 Virtual Memory

We implemented the new MMU instructions and run-time checks described in Section 5.5.5 and ported the SVA Linux kernel to use the new instructions. Linux already contains macros to allocate, modify and free page table pages. We modified these macros to use our new API (which is based on the `paravirtops` interface from Linux 2.6). We implemented all of the run-time checks except for those that ensure that I/O

device memory isn't mapped into kernel memory objects. These checks require that the kernel allocate all I/O memory objects within a predefined range of the virtual address space, which our Linux kernel does not currently do.

5.7 Evaluation and Analysis

Our evaluation has two goals. First, we wanted to determine whether our design for low-level software/hardware interaction was effective at stopping security vulnerabilities in commodity OS kernels. Second, we wanted to determine how much overhead our design would add to an already existing memory-safety system.

5.7.1 Exploit Detection

We performed three experiments to verify that our system catches low-level hardware/software errors: First, we tried two different exploits on our system that were reported on Linux 2.4.22, the Linux version that is ported to SVA. The exploits occur in the MMU subsystem; both give an attacker root privileges. Second, we studied the e1000e bug [45]. We could not duplicate the bug because it occurs in Linux 2.6, but we explain why our design would have caught the bug if Linux 2.6 had been ported to SVA. Third, we inserted many low-level operation errors inside the kernel to evaluate whether our design prevents the safety violations identified in Section 5.2.

Linux 2.4.22 exploits. We have identified three reported errors for Linux 2.4.22 caused by low-level kernel-hardware interactions [23, 134, 138]. Our experiment is limited to these errors because we needed hardware/software interaction bugs that were in Linux 2.4.22. Of these, we could not reproduce one bug due to a lack of information in the bug report [23]. The other two errors occur in the `mremap` system call but are distinct errors.

The first exploit [138] is due to an overflow in a count of the number of times a page is mapped. The exploit code overflows the counter by calling `fork`, `mmap`, and `mremap` a large number of times. It then releases the page, giving it back to the kernel. However, the exploit code still has a reference to the page; therefore, if the page is reallocated for kernel use, the exploit code can read and modify kernel data. Our system catches this error because it disallows allocating kernel objects in a physical page mapped in user space.

The second exploit [134] occurs because of a missing error check in `mremap` which causes the kernel to place page table pages with valid page table entries into the page table cache. However, the kernel assumes

that page table pages in the page table cache do not contain any entries. The exploit uses this vulnerability by calling `mmap, mremap` and `munmap` to release a page table page with page entries that contain executable memory. Then, on an `exec` system call, the linker, which executes with root privileges, allocates a page table page, which happens to be the previously released page. The end result is that the linker jumps to the exploit's executable memory and executes the exploit code with root privileges. The SVA-M VM prevents this exploit by always zeroing page table pages when they are placed in a page directory so that no new, unintended, memory mappings are created for existing objects.

The e1000e bug. The fundamental cause of the e1000e bug is a memory load/store (the x86 `cmpxchg` instruction) on a dangling pointer, which happens to point to an I/O object. The `cmpxchg` instruction has non-deterministic behavior on I/O device memory and may corrupt the hardware. The instruction was executed by the `ftrace` subsystem, which uses self-modifying code to trace the kernel execution. It took many weeks for skilled engineers to track the problem. With our new safety checks, SVA-M would have detected the bug at its first occurrence. The self-modifying code interface of SVA-OS only allows enabling and disabling of code; writes to what the kernel (incorrectly) thought was its code is not possible. SVA-M actually has a second line of defense if (hypothetically) the self-modifying code interface did not detect it: SVA-M would have prevented the I/O memory from being mapped into code pages, and thus prevented this corruption. (And, hypothetically again, if a dangling pointer to a data object had caused the bug, SVA-M would have detected any ordinary reads and writes trying to write to I/O memory locations.)

Kernel error injection. To inject errors, we added new system calls into the kernel; each system call triggers a specific kind of kernel/hardware interaction error that either corrupts memory or alters control flow. We inserted four different errors. The first error modifies the saved Integer State of a process so that an invalid Integer State is loaded when the process is scheduled. The second error creates a new MMU mapping of a page containing type-known kernel memory objects and modifies the contents of the page. The third error modifies the MMU mappings of pages in the stack range. The fourth error modifies the internal metadata of SVA-M to set incorrect bounds for all objects. This last error shows that with the original design, we can *disable the SVA-M memory safety checks that prevent Linux exploits*; in fact, it would not be difficult to do so with this bug alone for three of the four kernel exploits otherwise prevented by the original SVA-M system in Chapter 4.

All of the injected errors were caught by the new SVA-M implementation. With the previous implementation, these errors either crash the kernel or create undefined behavior. This gives us confidence about

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)	% Increase from i386 to SVA-OS	Description
bzip2	18.7 (0.47)	18.3 (0.47)	18.0 (0.00)	0.0%	Compressing 64 MB file
lame	133.3 (3.3)	132 (0.82)	126.0 (0.82)	-0.1%	Converting 206 MB WAV file to MP3
perl	22.3 (0.47)	22.3 (0.47)	22.3 (0.47)	0.0%	Interpreting scrabbl.pl from SPEC 2000

Table 5.3: Latency of Applications. Standard Deviation Shown in Parentheses.

the correctness of our new design and implementation of SVA-M. Note that we only injected errors that our design addresses because we believe that our design is "complete" in terms of the possible errors due to kernel-hardware interactions. Nevertheless, the injection experiments are useful because they *validate that the design and implementation actually solve these problems.*

5.7.2 Performance

To determine the impact of the additional run-time checks on system performance, we ran several experiments with applications typically used on server and end-user systems. We ran tests on the original Linux 2.4.22 kernel (marked i386 in the figures and tables), the same kernel with the original SVA-M safety checks from Chapter 4 (marked SVA), and the SVA kernel with our safety checks for low-level software/hardware interactions (marked SVA-OS).

It is important to note that an underlying memory safety system like SVA-M can incur significant run-time overhead for C code, especially for a commodity kernel like Linux that was not designed for enforcement of memory safety. Such a system is not the focus of this chapter. Although we present our results relative to the original (unmodified) Linux/i386 system for clarity, we focus the discussion on the excess overheads introduced by SVA-OS beyond those of SVA-M since the new techniques in SVA-OS are the subject of this chapter.

We ran these experiments on a dual-processor AMD Athlon 2100+ at 1,733 MHz with 1 GB of RAM and a 1 Gb/s network card. We configured the kernel as an SMP kernel but ran it in on a single processor since the SVA-M implementation is not yet SMP safe. Network experiments used a dedicated 1 Gb/s switch. We ran our experiments in single-user mode to prevent standard system services from adding noise to our performance numbers.

We used several benchmarks in our experiments: the thttpd Web server, the OpenSSH sshd encrypted file transfer service, and three local applications – bzip2 for file compression, the lame MP3 encoder, and a perl

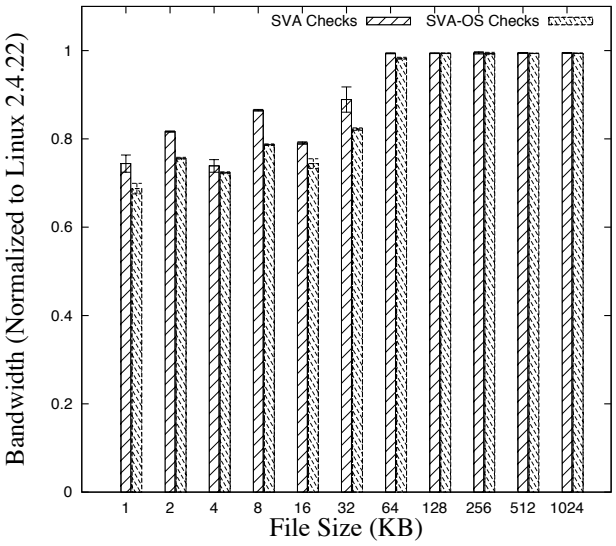


Figure 5.1: Web Server Bandwidth (Linux/i386 = 1.0)

Benchmark	i386 (s)	SVA (s)	SVA-OS (s)
bzip2	41	40	40
lame	203	202	202
perl	24	23	23

Table 5.4: Latency of Applications During Priming Run.

interpreter. These programs have a range of different demands on kernel operations. Finally, to understand why some programs incur overhead while others do not, we used a set of microbenchmarks including the HBench-OS microbenchmark suite [31] and two new tests we wrote for the poll and select system calls.

Application Performance First, we used ApacheBench to measure the file-transfer bandwidth of the thttpd web server [115] serving static HTML pages. We configured ApacheBench to make 5,000 requests using 25 simultaneous connections. Figure 5.1 shows the results of both the original SVA kernel and the SVA kernel with the new run-time checks described in Section 5.5. Each bar is the average bandwidth of 3 runs of the experiment; the results are normalized to the original i386 Linux kernel. For small files (1 KB - 32 KB) in which the original SVA-M system adds significant overhead, our new run-time checks incur a small amount of additional overhead (roughly a 9% decrease in bandwidth relative to the original SVA-M system). However, for larger file sizes (64 KB or more), the SVA-OS checks add negligible overhead to the original SVA-M system.

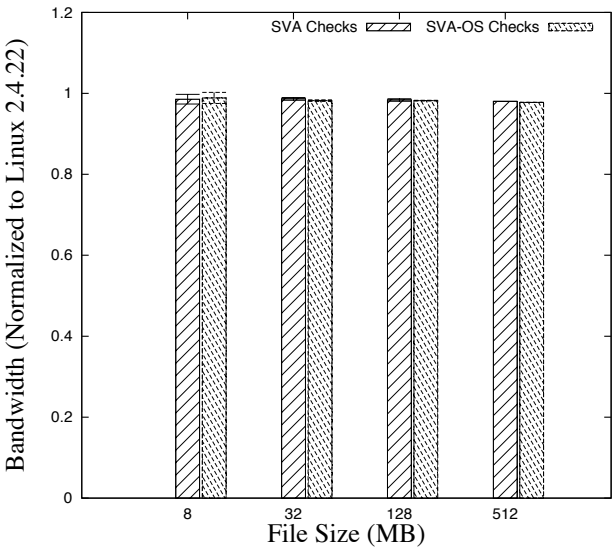


Figure 5.2: SSH Server Bandwidth (Linux/i386 = 1.0)

We also measured the performance of `sshd`, a login server offering encrypted file transfer. For this test, we measured the bandwidth of transferring several large files from the server to our test client; the results are shown in Figure 5.2. For each file size, we first did a priming run to bring file system data into the kernel's buffer cache; subsequently, we transferred the file three times. Figure 5.2 shows the mean of the receive bandwidth of the three runs normalized to the mean receive bandwidth measured on the original i386 kernel; note that the units on the X-axis are MB. Our results indicate that there is no significant decrease in bandwidth due to the extra run-time checks added by the original SVA-M system or the new run-time checks presented in this chapter. This outcome is far better than thttpd, most likely due to the large file sizes we transferred via `scp`. For large file sizes, the network becomes the bottleneck: transferring an 8 MB file takes 62.5 ms on a Gigabit network, but the overheads for basic system calls (shown in Table 5.5) show overheads of only tens of microseconds.

To see what effect our system would have on end-user application performance, we ran experiments on the client-side programs listed in Table 5.3. We tested bzip2 compressing a 64 MB file, the LAME MP3 encoder converting a 206 MB file from WAV to MP3 format, and the perl interpreter running the training input from the SPEC 2000 benchmark suite. For each test, we ran the program once to prime any caches within the operating system and then ran each program three times. Table 5.3 shows the average of the execution times of the three runs and the percent overhead that the applications experienced executing on the new SVA-M kernel (labeled "SVA-OS" in the table) relative to the original i386 Linux kernel. The results

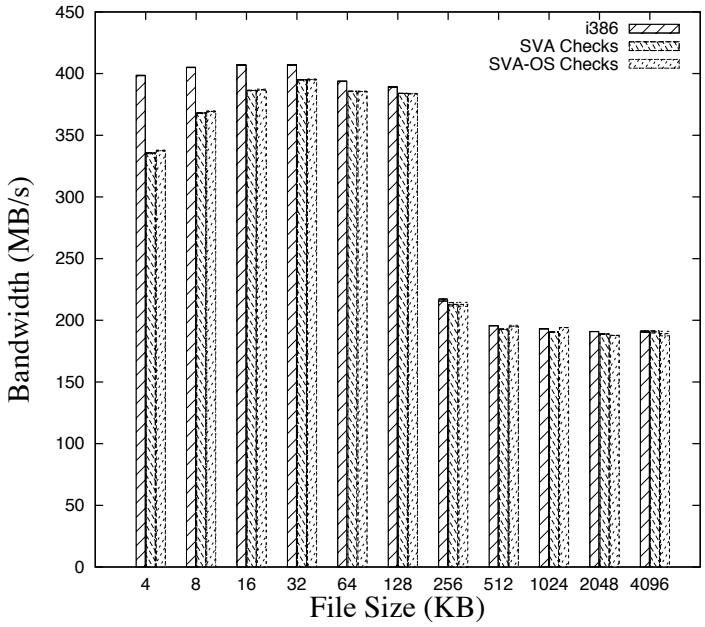


Figure 5.3: File System Bandwidth

show that our system adds virtually no overhead for these applications, even though some of the programs (bzip2 and lame) perform substantial amounts of I/O. Table 5.4 shows the latency of the applications during their priming runs; our kernel shows no overhead even when the kernel must initiate I/O to retrieve data off the physical disk.

Microbenchmark Performance To better understand the different performance behaviors of the applications, we used microbenchmarks to measure the overhead our system introduces for primitive kernel operations. For these experiments, we configured HBench-OS to run each test 50 times.

Our results for basic system calls (Table 5.5) indicate that the original SVA-M system adds significant overhead (on the order of tens of microseconds) to individual system calls. However, the results also show that our new safety checks only add a small amount of additional overhead (25% or less) to the original SVA-M system.

We also tested the file system bandwidth, shown in Figure 5.3. The results show that the original SVA-M system reduces file system bandwidth by about 5-20% for small files but that the overhead for larger files is negligible. Again, however, the additional checks for low-level kernel operations add no overhead.

The microbenchmark results provide a partial explanation for the application performance results. The applications in Table 5.3 experience no overhead because they perform most of their processing in user-space;

Benchmark	i386 (μs)	SVA (μs)	SVA-OS (μs)	% Increase from SVA to SVA-OS	Description
getpid	0.16 (0.001)	0.37 (0.000)	0.37 (0.006)	0.0%	Latency of getpid() syscall
openclose	1.10 (0.009)	11.1 (0.027)	12.1 (0.076)	9.0%	Latency of opening and closing a file
write	0.25 (0.001)	1.87 (0.012)	1.86 (0.010)	-0.4%	Latency of writing a single byte to /dev/null
signal handler	1.59 (0.006)	6.88 (0.044)	8.49 (0.074)	23%	Latency of calling a signal handler
signal install	0.34 (0.001)	1.56 (0.019)	1.95 (0.007)	25%	Latency of installing a signal handler
pipe latency	2.74 (0.014)	30.5 (0.188)	35.9 (0.267)	18%	Latency of ping-ponging one byte message between two processes
poll	1.16 (0.043)	6.47 (0.080)	7.03 (0.014)	8.7%	Latency of polling both ends of a pipe for reading and writing. Data is always available for reading.
select	1.00 (0.019)	8.18 (0.133)	8.81 (0.020)	7.7%	Latency of testing both ends of a pipe for reading and writing. Data is always available for reading.

Table 5.5: Latency of Kernel Operations. Standard Deviation Shown in Parentheses.

the overhead of the kernel does not affect them much. In contrast, the `sshd` and thttpd servers spend most of their time executing in the kernel (primarily in the poll(), select(), and write() system calls). For the system calls that we tested, our new safety checks add less than several microseconds of overhead (as shown in Table 5.5). For a small network transfer of 1 KB (which takes less than 8 μs on a Gigabit network), such an overhead can affect performance. However, for larger files sizes (e.g., an 8 MB transfer that takes 62.5 ms), this overhead becomes negligible. This effect shows up in our results for networked applications (thttpd and sshd): smaller file transfers see significant overhead, but past a certain file size, the overhead from the run-time safety checks becomes negligible.

5.8 Related Work

Previous work has explored several approaches to providing greater safety and reliability for operating system kernels. Some require complete OS re-design, e.g., capability-based operating systems [128, 129] and microkernels [11, 92]. Others use isolation (or "sandboxing") techniques, including device driver isolation within the OS [125, 142, 144, 164] or the hypervisor [66]. While effective at increasing system reliability,

none of these approaches provide the memory safety guarantees provided by our system, e.g., none of these prevent corruption of memory mapped I/O devices, unsafe context switching, or improper configuration of the MMU by either kernel or device driver code. In fact, none of these approaches could protect against the Linux exploits or device corruption cases described in Section 5.7. In contrast, our system offers protection from all of these problems for both driver code and core kernel code.

The EROS [129] and Coyotos [128] systems provide a form of safe (dynamic) typing for abstractions, e.g., capabilities, at their higher-level OS ("node and page") layer. This type safety is preserved throughout the design, even across I/O operations. The lower-level layer, which implements these abstractions, is written in C/C++ and is theoretically vulnerable to memory safety errors but is designed carefully to minimize them. The design techniques used by these systems are extremely valuable but difficult to retrofit to commodity systems.

Some OSs written in type-safe languages, including JX [70], SPIN [76], Singularity [78], and others [73] provide abstractions that guarantee that loads and stores to I/O devices do not access main memory, and main memory accesses do not access I/O device memory. However, these systems either place context switching and MMU management within the virtual machine run-time (JX) or provide no guarantee that errors in these operations cannot compromise the safety guarantees of the language in which they are written.

Verve [156] is an operating system written in C# with a small software layer called Nucleus that is written in a language called Boogie. The C# portions of the kernel are compiled into typed assembly language (TAL) [106] so that the type-safety present at the C# source-code level can be verified to hold in the generated machine code. Nucleus (which handles operations such as context switching and memory allocation) is written with proof annotations that are used by an SMT solver to mechanically verify that invariants necessary for memory and type safety hold. Unlike Verve, SVA-M can support operating systems written in type-unsafe languages like C and does not rely upon garbage collection to enforce its safety guarantees. SVA-M's low-level hardware instructions are also more complete than those found in Nucleus; for example, Nucleus does not currently support general I/O or MMU reconfiguration as SVA-M does. Unlike SVA-M, Verve's and Nucleus's *implementation* is verified to maintain memory and type safety whereas SVA-M assumes that its low-level instructions are implemented correctly.

Another approach that could provide some of the guarantees of our work is to add annotations to the C language. For example, SafeDrive's annotation system [164] could be extended to provide our I/O memory protections and perhaps some of our other safety guarantees. Such an approach, however, would likely require changes to every driver and kernel module, whereas our approach only requires a one-time port to

the SVA instruction set and very minor changes to machine-independent parts of the kernel.

The Devil project [101] defines a safe interface to hardware devices that enforces safety properties. Devil could ensure that writes to the device's memory did not access kernel memory, but not vice versa. Our SVA-M extensions also protect I/O memory from kernel memory and provide comprehensive protection for other low-level hardware interactions, such as MMU changes, context switching, and thread management.

Mondrix [154] provides isolation between memory spaces within a kernel using a word-granularity memory isolation scheme implemented in hardware [153]. Because Mondrix enables much more fine-grained isolation (with acceptable overhead) than the software supported isolation schemes discussed earlier, it may be able to prevent some or all of the memory-related exploits we discuss. Nevertheless, it cannot protect against other errors such as control flow violations or stack manipulation.

A number of systems provide Dynamic Information Flow Tracking or "taint tracking" to enforce a wide range of security policies, including memory safety, but most of these have only reported results for user-space applications. Raksha [54] employed fine-grain information flow policies, supported by special hardware, to prevent buffer overflow attacks on the Linux kernel by ensuring that injected values weren't dereferenced as pointers. Unlike our work, it does not protect against attacks that inject non-pointer data nor does it prevent use-after-free errors of kernel stacks and other state buffers used in low-level kernel/hardware interaction. Furthermore, this system does not work on commodity hardware.

The CacheKernel [39] partitions its functionality into an application-specific OS layer and a common "cache kernel" that handles context-switching, memory mappings, etc. The CacheKernel does not aim to provide memory safety, but its two layers are conceptually similar to the commodity OS and the virtual machine in our approach. A key design difference, however, is that our interface also attempts to make kernel code easier to analyze. For example, state manipulation for interrupted programs is no longer an arbitrary set of loads/stores to memory but a single instruction with a semantic meaning.

Our system employs techniques from VMMs. The API provided by SVA for configuring the MMU securely is similar to that presented by para-virtualized hypervisors [60, 155]. However, unlike VMMs, our use of these mechanisms is to provide fine-grain protection internal to a single domain, including isolation between user and kernel space and protection of type-safe main memory, saved processor state, and the kernel stack.

We believe VMMs could be a useful *target* for our work because they can be susceptible to the same attacks as operating system kernels. VMMs are relatively large systems written in C that interface directly with the hardware. Code injection attacks already exist for hypervisors [149]; there is no fundamental reason

why other low-level attacks are not possible. Current hypervisors would not be able to guard against [138], which our system does prevent, even though it is an MMU error. A hypervisor that uses binary rewriting internally, e.g., for instrumenting itself, could be vulnerable to [45], just as the Linux kernel was. SVA-M could protect VMMs from these types of bugs just as it does for the Linux kernel.

SecVisor [126] is a hypervisor that ensures that only approved code is executed in the processor's privileged mode. In contrast, our system does not ensure that kernel code meets a set of requirements other than being memory safe. Unlike SVA-M, SecVisor does not ensure that the approved kernel code is memory safe.

5.9 Summary

We have presented new mechanisms to ensure that low-level kernel operations such as processor state manipulation, stack management, memory mapped I/O, MMU updates, and self-modifying code do not violate the assumptions made by memory safety checkers. We implemented our design in the SVA-M system which provides a safe execution environment for commodity operating systems and its corresponding port of Linux 2.4.22. Only around 100 lines of code were added or changed to the SVA-ported Linux kernel for the new techniques. To our knowledge, this is the first paper that (i) describes a design to prevent bugs in low-level kernel operations from compromising memory safe operating systems, including operating systems written in safe or unsafe languages; and (ii) implements and evaluates a system that guards against such errors.

Our experiments show that the additional runtime checks add little overhead to the original SVA-M prototype and were able to catch multiple real-world exploits that would otherwise bypass the memory safety guarantees provided by the original SVA-M system. Taken together, these results indicate that it is clearly worthwhile to add these techniques to an existing memory safety system.

Chapter 6

Control-Flow Integrity for Operating System Kernels

6.1 Introduction

Many memory safety attacks work by diverting a program's control flow to instructions of the attacker's choosing; these instructions may be injected by the attacker [17] or may already exist within the program [132, 118]. Control-flow integrity (CFI) is a family of security policies that thwart such attacks. Early CFI policies require that all computed branches (e.g., returns from functions and indirect function calls) jump to virtual addresses that are designated as correct via static analysis [10]. Additional restrictions to CFI, such as those imposed by SVA-M (Chapter 5) and HyperSafe [149], require that the instructions within the code segment do not change.

Enforcing CFI on commodity operating system kernel code could provide protection against control hijack attacks that is comprehensive, efficient, and straightforward to implement. However, operating systems pose three challenges for existing CFI techniques. First, not all targets of indirect control transfers can be determined statically from the kernel code. Interrupts can occur at *any* instruction boundary, so the kernel must be able to transfer control to any interrupted instruction on a return from interrupt. Second, operating system operations affect control flow in complicated ways. Signal handler dispatch, for example, modifies the program counter in interrupted program state saved in memory [30, 99], and efficient user-kernel memory copying functions modify interrupted kernel state [30] to recover from page protection faults. Third, operating systems have access to privileged hardware that invalidate assumptions commonly made by CFI techniques. As an example, some CFI systems [10, 159] assume that the code segment is non-writable. Errant DMA and MMU configurations can invalidate that assumption as discussed in previous work [149] as well as in Chapter 5.

Most solutions for enforcing CFI [10, 162, 159] do not protect commodity operating system code. The few that do protect system-level code have serious limitations: HyperSafe [149] only protects a hypervisor and does not provide control-flow integrity for operations found in operating systems (e.g., signal handler

dispatch); it also does not protect against return to user (ret2usr) attacks [82] that corrupt the program counter saved on interrupts, traps, and system calls to execute code belonging to less-privileged software. The kGuard [82] system, designed to thwart ret2usr attacks, enforces a very weak CFI variant that only ensures that control-flow is directed at virtual addresses within the kernel; some of its protection is probabilistic, and it does not handle attacks that use the MMU to change the instructions within the code segment. Secure Virtual Architecture for memory safety (SVA-M) (Chapters 4 and 5) provides comprehensive protection against control hijacking attacks, but it does so with heavyweight memory-safety techniques that have relatively high overheads even after being optimized by techniques using sophisticated whole-program pointer analysis [88].

Furthermore, very few CFI systems formally prove that they enforce control-flow integrity; those that do [10, 163] do not model one or more features used by operating system kernels, such as virtual memory, trap handlers, context switching, and signal delivery. Having an approach for enforcing control-flow integrity on these operations that has been formally verified would increase confidence that the approach works correctly.

We have built a system named KCoFI (Kernel Control Flow Integrity, pronounced "coffee") that aims to provide comprehensive, efficient, and simple protection against control flow attacks for a complete commodity operating system. KCoFI operates between the software stack and processor. Essentially, KCoFI uses traditional label-based protection for programmed indirect jumps [10] but adds a thin run-time layer linked into the OS that protects some key OS data structures like thread stacks and monitors all low-level state manipulations performed by the OS. KCoFI is built using SVA (described in Chapter 3) which provides the compiler capabilities and the ability to monitor low-level state manipulation. Our system provides the first comprehensive control-flow integrity enforcement for commodity OS kernels that does not rely on slower and more sophisticated memory safety techniques. Our protection thwarts both classical control flow attacks as well as ret2usr attacks. To verify that our design correctly enforces control-flow integrity, we have built a formal model of key features of our system (including the new protections for OS operations) using small-step semantics and provided a proof that our design enforces control-flow integrity. The proofs are encoded in the Coq proof assistant and are mechanically verified by Coq.

The contributions of this chapter are as follows:

- We provide the first complete control-flow integrity solution for commodity operating systems that does not rely on sophisticated whole-program analysis or a much stronger and more expensive security policy like complete memory safety.

- We have built a formal model of kernel execution with small-step semantics that supports virtual to physical address translation, trap handling, context switching, and signal handler dispatch. We use the model to provide a proof that our design prevents CFI violations (we do not verify our implementation).
- We evaluate the security of our system for the FreeBSD 9.0 kernel on the x86-64 architecture. We find that all the Return Oriented Programming (ROP) gadgets found by the ROPGadget tool [121] become unusable as branch targets. We also find that our system reduces the average number of possible indirect branch targets by 98.18%.
- We evaluate the performance of our system and find that KCoFI has far lower overheads than the SVA-M system presented in Chapters 4 and 5 (the only other system which provides full control-flow integrity to commodity OS kernels). Compared to an unmodified kernel, KCoFI has relatively low overheads for server benchmarks but higher overheads for an extremely file-system intensive benchmark.

The remainder of the chapter is organized as follows: Section 6.2 describes our attack model. Section 6.3 provides an overview of the KCoFI architecture. Section 6.4 presents the design of KCoFI and how it enforces control-flow integrity, and Section 6.5 presents an overview of our formal control-flow integrity proof. Section 6.6 describes our implementation while Section 6.7 evaluates its efficacy at thwarting attacks and Section 6.8 describes the performance of our system. Section 6.9 describes related work, and Section 6.10 concludes.

6.2 Attack Model

In our attack model, we assume that the OS is benign but may contain vulnerabilities; we also assume that the OS has been properly loaded without errors and is executing. Our model allows the attacker to trick the kernel into attempting to modify any memory location. We additionally assume that the attacker is using such corruption to modify control-data, including targets that are not of concern to traditional CFI techniques, e.g., processor state (including the PC and stack pointer) saved in memory after a context-switch; trap and interrupt handler tables; invalid pointer values in user-kernel copy operations; malicious MMU reconfiguration; etc. Non-control data attacks [37] are excluded from our model.

Notice that external attackers in our model can influence OS behavior only through system calls, I/O, and traps. For example, dynamically loaded device drivers are assumed not to be malicious, but may also be buggy (just like the rest of the OS kernel), and will be protected from external attack. We assume that the system is employing secure boot features such as those found in AEGIS [22] or UEFI [145] to ensure

that KCoFI and the kernel are not corrupted on disk and are the first pieces of software loaded on boot. We further assume that the attacker does not have physical access to the machine; hardware-based attacks are outside the scope of our model.

6.3 KCoFI Infrastructure

KCoFI has several unique requirements. First, it must instrument commodity OS kernel code; existing CFI enforcement mechanisms use either compiler or binary instrumentation [10, 162, 160]. Second, KCoFI must understand how and when OS kernel code interacts with the hardware. For example, it must understand when the OS is modifying hardware page tables in order to prevent errors like writeable and executable memory. Third, KCoFI must be able to control modification of interrupted program state in order to prevent ret2usr attacks.

The Secure Virtual Architecture (SVA) (Chapter 3) provides the infrastructure that KCoFI needs. Figure 6.1 shows how KCoFI adapts the SVA infrastructure to enforce control-flow integrity on system software (such as an operating system or hypervisor). All software, including the operating system and/or hypervisor, is compiled to the virtual instruction set that SVA provides. As described in Chapter 2, the SVA virtual machine (VM) translates code from the virtual instruction set to the native instruction set either ahead-of-time (by caching virtual instruction set translations) or just-in-time while the application is running.

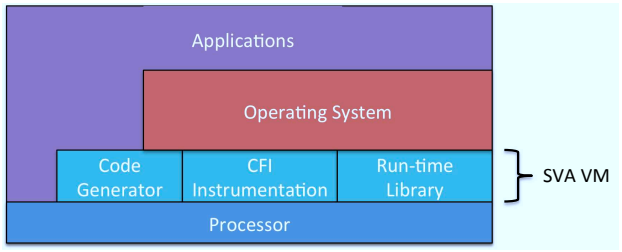


Figure 6.1: SVA/KCoFI Architecture

The SVA infrastructure enables KCoFI to enforce a CFI policy by using the SVA compiler instrumentation capabilities and using the SVA-OS instruction set to identify and control both OS kernel/hardware interactions and OS kernel/application interactions. The SVA compiler instrumentation capabilities permit KCoFI to control function returns and indirect function calls; it also allows KCoFI to protect access to data structures containing control data. The SVA-OS instructions permit KCoFI to control the MMU configuration and to detect control-flow integrity errors when the kernel performs state manipulation operations.

KCoFI requires all OS code, including kernel extensions, to be compiled to the virtual instruction set but allows applications to be compiled to either the virtual or native instruction set.

6.4 Design

In this section, we describe the CFI policy that KCoFI enforces and the hardware and compiler mechanisms it employs to enforce the policy.

6.4.1 Control-flow Integrity Policy and Approach

KCoFI enforces context-insensitive CFI like that of Abadi et. al. [10]: calls to functions must jump to the beginning of some function, and returns must jump back to one of the call sites that could have called the exiting function. The return address is not itself protected, so it is possible for a function to dynamically return to a call site other than the one that called the function in that specific execution.

To enforce CFI, Abadi et. al. [10] insert special byte sequences called *labels* at the targets of indirect control transfers within the code segment. These labels must not appear anywhere else within the instruction stream. Their technique then inserts code before indirect jumps to check that the address that is the target of the indirect jump contains the appropriate label. Abadi et. al. provided a formal proof that their technique enforces control-flow integrity if the code segment is immutable [10].

The KCoFI VM instruments the code with the needed labels and run-time checks when translating code from the virtual instruction set to the processor's native instruction set. To avoid complicated static analysis, KCoFI does not attempt to compute a call graph of the kernel. Instead, it simply labels all targets of indirect control transfers with a single label. Our design also uses a jump table optimization [160] to reduce the number of labels and CFI checks inserted for switch statements. While our current design effectively uses a very conservative call graph, note that a more sophisticated implementation that computes a more precise call graph can be made without changing the rest of the design. Also, the MMU protections (discussed in Section 6.4.3) ensure that the code segment is not modified by errant writes.

One issue with using CFI labels is that a malicious, native code user-space application could place CFI labels within its own code to trick the instrumentation into thinking that its code contains a valid kernel CFI target [82]. KCoFI solves this problem by adapting a technique from kGuard [82]; before checking a CFI label, it masks the upper bits of the address to force the address to be within the kernel's address space. This approach allows a kernel on the KCoFI system to run applications that are compiled directly to native code (i.e., not compiled to the virtual instruction set).

Similarly, the SVA-OS instructions described later in this section are implemented as a run-time library that is linked into the kernel. This run-time library is instrumented with a disjoint set of CFI labels for its internal functions and call sites to ensure that indirect branches in the kernel do not jump into the middle of the implementation of an SVA-OS instruction. In this way, the run-time checks that these library functions perform cannot be bypassed.

Name	**Description**
sva.declare.ptp (void * ptp, unsigned level)	Zeroes the physical page mapped to the direct map pointer *ptp* and marks it as a page table page at level *level*.
sva.remove.ptp (void * ptp)	Checks that the physical page pointed to by direct map pointer *ptp* is no longer used and marks it as a regular page.
sva.update.l1.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L1 page, validate that the translation *trans* does not violate any security policies and store it into *ptp*.
sva.update.l2.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L2 page, validate that the translation *trans* maps an L1 page and store *trans* into *ptp*.
sva.update.l3.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L3 page, validate that the translation *trans* maps an L2 page and store *trans* into *ptp*.
sva.update.l4.mapping (void * ptp, unsigned trans)	If *ptp* is a direct map pointer to an L4 page, validate that the translation *trans* maps an L3 page and store *trans* into *ptp*.
sva.load.pagetable (void * ptp)	Check that the physical page mapped to the direct map pointer *ptp* is an L4 page and, if so, make it the active page table.

Table 6.1: KCoFI MMU Instructions

6.4.2 Protecting KCoFI Memory with Software Fault Isolation

The original CFI technique of Abadi et al. [10] is stateless in that the only data used are constant labels embedded in the code segment of the application being protected, either as immediate operands to checking instructions or as constant labels at control transfer targets.[1] KCoFI, however, needs to maintain some additional state to protect privileged kernel behaviors, which do not occur in userspace code. This state includes hardware trap vector tables, page mapping information, interrupted program state (as described in Section 6.4.6), and other state that, if corrupted, could violate control-flow integrity. While the MMU can protect code memory because such memory should never be writeable, KCoFI will need to store this state in memory that can be written by KCoFI but not by errant operating system and application writes. KCoFI

[1] One variant of their design uses x86 segmentation registers to protect application stack frames containing return addresses.

uses lightweight instrumentation on kernel store instructions to protect this memory: essentially a version of software-fault isolation [148]. (An alternative would be to use MMU protections on KCoFI data pages as well, but that would incur additional numerous TLB flushes.)

As Figure 6.2 shows, our design calls for a reserved portion of the address space called KCoFI memory which will contain the KCoFI VM's internal memory. KCoFI uses the MMU to prevent user-space code from writing into KCoFI memory. To prevent access by the kernel, KCoFI instruments all instructions in the kernel that write to memory; this instrumentation uses simple bit-masking that moves pointers that point into KCoFI memory into a reserved region of the address space (marked "Reserved" in Figure 6.2). This reserved region can either be left unmapped so that errant writes are detected and reported, or it can have a single physical frame mapped to every virtual page within the region so that errant writes are silently ignored. Note that only stores need instrumentation; none of the KCoFI internal data needs to be hidden from the kernel, and, as a result, can be freely read by kernel code.

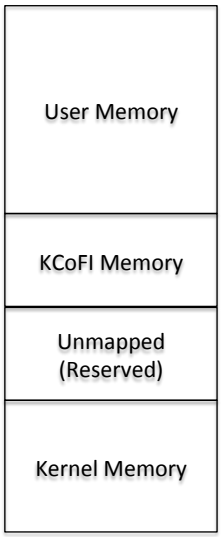


Figure 6.2: KCoFI Address Space Organization

6.4.3 MMU Restrictions

As Chapter 5 and the HyperSafe project have shown [149], MMU configuration errors can lead to violations of security policies enforced by inline reference monitors. As KCoFI's CFI mechanism must keep code read-only and the store instrumentation makes assumptions about the address space layout, KCoFI must be able to restrict how the operating system configures the MMU.

The SVA infrastructure forces hardware page table pages to be read-only and requires the OS to use special instructions to make changes to the page table pages (see Chapter 5). KCoFI simplifies and enhances the original SVA-OS MMU instructions; these instructions are shown in Table 6.1.

The KCoFI VM maintains a structure within its portion of the address space called the *direct map*. The direct map is a one-to-one mapping between virtual addresses within the direct map and physical frames of memory. All pages in the direct map will be read-only. The purpose of the direct map is to provide the KCoFI VM and the OS kernel with a known virtual address for every physical address. When the OS asks the KCoFI VM to make page table updates, it will identify page table pages by their direct map virtual address.

The `sva.declare.ptp()` instruction informs the KCoFI VM of which frames will be used for hardware page table pages and at which level of the page table hierarchy these frames will be used. The KCoFI VM will only permit pages that are not in use to be declared for use as page table pages, and it will zero the page's contents to ensure that no stale virtual to physical page translations remain in the page. When the system wishes to stop using a frame as a page table page, it can call `sva.remove.ptp()`. When called, `sva.remove.ptp()` verifies that the frame is no longer referenced by any page table pages; if the check passes, it allows the frame to be mapped read/write into the virtual address space and used for kernel or user-space data.

The `sva.update.l<n>.mapping()` instructions write a new page table entry into a page table page previously declared using the `sva.declare.ptp()` instruction. The KCoFI VM will first vet the mapping before inserting it into the page table page at the specified offset. For example, if the mapping should insert an L2 page table page, the checks ensure that the physical address specified in the translation is a page previously declared as an L2 page. The instructions will also keep count of how many references there are to each physical page frame.

Switching from one page table to another is done by the `sva.load.pagetable()` instruction which requires that it be given the address of a level 4 page table page.

There are two ways in which reconfiguration of the MMU can allow the operating system to bypass the protections provided by the compiler instrumentation. First, an errant operating system may reconfigure the virtual addresses used for KCoFI memory or the code segment so that they either permit write access to read-only data or map new physical frames to the virtual pages, thereby modifying their contents. Second, an errant operating system might create new virtual page mappings in another part of the address space to physical pages that are mapped into KCoFI memory or the code segment. Since the CFI and store

Name	Description
sva.icontext.save (void)	Push a copy of the most recently created Interrupt Context on to the thread's Saved Interrupt Stack within the KCoFI VM internal memory.
sva.icontext.load (void)	Pop the most recently saved Interrupt Context from the thread's Saved Interrupt Context stack and use it to replace the most recently created Interrupt Context on the Interrupt Stack.
sva.ipush.function (int (*f)(...), ...)	Modify the state of the most recently created Interrupt Context so that function *f* has been called with the given arguments. Used for signal handler dispatch.
sva.init.icontext (void * stackp, unsigned len, int (*f) (...), ...)	Create a new Interrupt Context with its stack pointer set to *stackp + len*. Also create a new thread that can be swapped on to the CPU and return its identifier; this thread will begin execution in the function *f*. Used for creating new kernel threads, application threads, and processes.
sva.reinit.icontext (int (*f) (...), void * stackp, unsigned len)	Reinitialize an Interrupt Context so that it represents user-space state. On a return from interrupt, control will be transferred to the function *f*, and the stack pointer will be set to *stackp*.

Table 6.2: KCoFI Interrupt Context Instructions

instrumentation makes assumptions about the location of the code segment and KCoFI memory within the address space, the MMU instructions must ensure that those assumptions hold. If these assumptions were to be broken, then both the code segment and KCoFI's internal data structures could be modified, violating control-flow integrity.

The KCoFI MMU instructions enforce the following restrictions on MMU configuration in order to protect the native code generated by the KCoFI VM:

1. No virtual addresses permitting write access can be mapped to frames containing native code translations.

2. The OS cannot create additional translations mapping virtual pages to native code frames.

3. Translations for virtual addresses used for the code segment cannot be modified.

Additional restrictions prevent the operating system from using the MMU to bypass the instrumentation on store instructions:

1. Translations for virtual addresses in KCoFI memory cannot be created, removed, or modified.

2. Translations involving the physical frames used to store data in KCoFI memory cannot be added, removed, or modified.

6.4.4 DMA and I/O Restrictions

Memory writes issued by the CPU are not the only memory writes that can corrupt the code segment or internal KCoFI memory. I/O writes to memory-mapped devices and external DMA devices can also modify memory. The KCoFI VM must control these memory accesses also.

KCoFI, like the original SVA-M system in Chapter 5, uses an I/O MMU to prevent DMA operations from overwriting the OS kernel's code segment, the KCoFI memory, and frames that have been declared as page table pages.

Protecting KCoFI memory from I/O writes is identical to the instrumentation for memory writes; pointers are masked before dereference to ensure that they do not point into the KCoFI memory. Additionally, the KCoFI VM prevents reconfiguration of the I/O MMU. KCoFI instruments I/O port writes to prevent reconfiguration for I/O MMUs configured using I/O port writes; memory-mapped I/O MMUs are protected using the MMU. The KCoFI VM can therefore vet configuration changes to the I/O MMU like it does for the MMU.

6.4.5 Thread State

The KCoFI VM provides a minimal thread abstraction for representing the processor state. This structure is called a thread structure and is referenced by a unique identifier. Internally, as shown in Figure 6.3, a thread structure contains the state of the thread when it was last running on the CPU (including its program counter) and two stacks of Interrupt Contexts (described in Section 6.4.6).

Thread structures are stored within the KCoFI memory to prevent direct modification by application or OS kernel code. The next few subsections will describe how the thread structure and the KCoFI instructions that manipulate them are used to provide interrupt, thread creation, and context switching operations that cannot violate control-flow integrity.

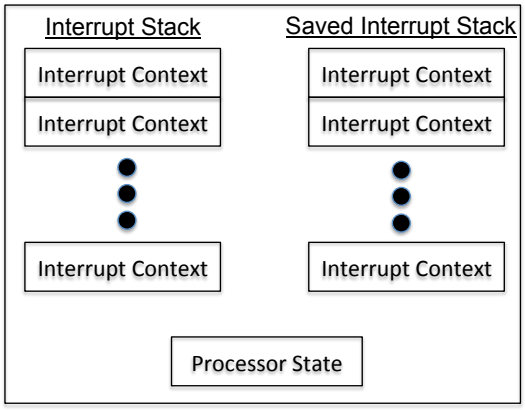


Figure 6.3: KCoFI Thread Structure

6.4.6 Protecting Interrupted Program State

When an interrupt, trap, or system call occurs, both the Linux and BSD operating systems store the interrupted program's state on the kernel stack [30, 99]. This state includes the return address at which execution should resume when the OS kernel decides to return from the interrupt, trap, or system call. Since it is stored in memory, this program counter value is vulnerable to corruption by memory safety errors.

Unlike other targets of indirect control transfers, the return address for a return-from-interrupt cannot be usefully determined via static analysis. Interrupts are allowed to occur at any time; any valid instruction location, in both application and kernel code, is permitted to be a valid return-from-interrupt target. The memory holding the return address must therefore be protected from corruption.

KCoFI saves the interrupted program state, called the *Interrupt Context*, on the Interrupt Context stack within the currently active thread's thread structure within the KCoFI memory. KCoFI then switches the stack pointer to a pre-determined kernel stack and transfers control to the OS kernel. Since the thread structure and stack of Interrupt Contexts are stored in KCoFI memory, the same bit-masking instrumentation used to protect the KCoFI memory also protects the return address for interrupts, traps, and system calls.

OS kernels need to make controlled modifications to interrupted program state in order to dispatch signal handlers [30, 99], efficiently recover from page faults when copying data between user and kernel space [30], or restart interrupted system calls [30]. The SVA infrastructure provides instructions for making such controlled changes (see Chapter 5); KCoFI provides new implementations of these instructions that do not rely on tracking memory object locations and sizes. These instructions are listed in Table 6.2.

The `sva.ipush.function()` instruction modifies interrupted program state to push a function call frame on to the interrupted program's stack; it is used for signal handler dispatch. Our design, like the original

SVA-M in Chapter 5, only permits this modification to be made to an Interrupt Context representing user-space state.

Signal handler dispatch uses `sva.icontext.save()` and `sva.icontext.load()` to save and restore interrupted program state before and after signal handler dispatch. The Saved Interrupt Stack is used to save a copy of an original interrupt context before the original (on the Interrupt Stack) is modified to dispatch a signal. In particular, the `sva.icontext.save()` instruction makes a copy of the Interrupt Context at the top of the Interrupt Stack and pushes this copy on to the Saved Interrupt Stack within the thread structure. The `sva.icontext.load()` instruction will pop an Interrupt Context off the Saved Interrupt Context stack and replace the top-most element on the Interrupt Stack with this previously saved Interrupt Context, ensuring that the correct state is resumed on the next return from interrupt. Unlike `sva.icontext.save()`, we restrict `sva.icontext.load()` so that it can only load user-space interrupted program state back into the Interrupt Context (since signals in a commodity kernel are generally never dispatched to interrupted kernel code, only to userspace code).

Name	**Description**
sva.swap (unsigned newID, unsigned * oldID)	Save the current processor native state and store an identifier representing it into *oldID* and then load the state represented by *newID*.

Table 6.3: KCoFI Context Switching Instructions

Name	**Description**
sva.translate (void * buffer, char * funcname, bool kmode)	Translate the SVA bitcode starting at buffer into native code. If *kmode* is true, then native code is generated for use in the processor's privileged mode. Otherwise, native code will be generated for use in the processor's unprivileged mode. The address to the function *funcname* will be returned.
sva.disable.privcode (void)	Disable further translation of SVA bitcode for use as native code in the processor's privileged state.

Table 6.4: KCoFI Native Code Translation Instructions

Exception handling within the kernel is done using the LLVM `invoke` and `unwind` instructions. The `invoke` instruction is just a `call` instruction with an additional label to identify an exception handler. `invoke` transfers control flow to the called function; if that function (or one of its callees) throws an exception, it uses the `unwind` instruction to unwind control-flow on the stack to the most recently executed `invoke` instruction [86].

The `sva.iunwind` instruction can modify interrupted privileged state; it is equivalent to forcing the interrupted program to execute an `unwind` instruction. This behavior cannot cause control flow to deviate from the compiler's precomputed call graph and is therefore safe to use.

6.4.7 Thread Creation

When a commodity operating system creates threads, it performs two tasks that can affect control flow. First, it allocates and initializes a kernel stack and places data on the new stack to contain the state that, when restored on a return from system call, will start running the new user-space thread [30]. Second, it creates new kernel state that will be placed on to the processor on a context switch [30]; after the context switch, the new state will return from the system call, loading the new interrupted program state and returning back to the application.

KCoFI provides the `sva.init.icontext()` instruction for creating new threads and processes. This instruction first creates a new thread structure which can be swapped on to the CPU using the `sva.swap()` instruction discussed in Section 6.4.8. This native processor state within the new thread structure is initialized so that it will begin execution in the function passed to `sva.init.icontext()`; the supplied function pointer is checked to ensure that it points to the beginning of a function.

The `sva.init.icontext()` instruction also creates empty Interrupt and Saved Interrupt stacks within the new thread structure. It then creates a new Interrupt Context that is identical to the top-most Interrupt Context in the current thread's Interrupt Stack; it then pushes this new Interrupt Context on to the top of the Interrupt Stack in the new thread structure. This new Interrupt Context is then modified to use the stack specified in the call to `sva.init.icontext()`.

Finally, `sva.init.icontext()` verifies that the specified stack does not overlap with KCoFI memory. If the check passes, it initializes the new stack so that a return from the specified function will return into the KCoFI VM system call dispatching code. The configuration of the Interrupt Context will ensure that if the function returns that control-flow integrity is not violated. When the function returns, control flow will return to the KCoFI VM which will attempt to return from a system call, trap, or interrupt. If the new Interrupt Context was cloned from the initial Interrupt Context from the first thread executing at system boot, the Interrupt Context will have a program counter value of zero and will therefore fault, preventing a CFI violation. Otherwise, the new Interrupt Context will have a valid program counter value from the Interrupt Context from which it was duplicated, and therefore, the return from interrupt will succeed.

6.4.8 Context Switching

Context switching requires saving the current processor state into memory and loading new state on to the processor. The state, including the stack pointer and program counter, are vulnerable while residing in memory.

As Table 6.3 shows, KCoFI provides an instruction called `sva.swap()` that saves the current processor state into the thread structure within KCoFI memory and loads new state that has been saved by a previous call to `sva.swap()` or created by `sva.init.icontext()`. State is represented by opaque identifiers returned by the `sva.swap()` and `sva.init.icontext()` instructions. This prevents the `sva.swap()` instruction from loading invalid program state. By saving state inside the KCoFI VM memory, the program counter within the saved state cannot be corrupted by memory safety errors. The `sva.swap()` instruction disables interrupts while it is executing, so that it cannot be interrupted and will never load inconsistent state.

The original SVA-M provides a similar instruction called `sva.swap.integer()` (see Chapter 5). The primary difference between the SVA-M instruction and the KCoFI version is that KCoFI does not split the native processor state into individual components; it saves integer registers, floating point registers, and vector registers. While not necessary for control-flow integrity, it does ensure that the correct floating point and vector state is restored on context switching, providing applications with a more secure context switch.

6.4.9 Code Translation

Any OS code (e.g., the core kernel or a driver) to be loaded for execution must start out in SVA bitcode form, whereas a user program can be SVA bitcode or native code. When the OS needs to load and execute any piece of software, it first passes the code to the `sva.translate` intrinsic shown in Table 6.4. The intrinsic takes a Boolean argument indicating whether the code should be translated for use in user-space or kernel-space mode. If this flag is true, the intrinsic verifies that the code is in SVA bitcode form. If the code is SVA bitcode, `sva.translate` will translate the bitcode into native code and cache it offline for future use. `sva.translate` returns a pointer to the native code of function `funcname`.

If the function pointer points to kernel code, the kernel can call the function directly; this permits the use of dynamically loaded kernel modules. If the function pointer points to user-mode code, then the kernel must use the `sva.reinit.icontext()` instruction to set up a user-space Interrupt Context that will begin execution of the application code when the Interrupt Context is loaded on to the processor on the next return from interrupt. These mechanisms provide a way of securely implementing the `exec()` family of system calls.

While KCoFI already prevents traditional native code injection (because the KCoFI VM prevents bad control-transfers and disallows executable and writable memory), it must also prevent *virtual* code injection attacks. A virtual code injection attack uses a memory safety error to modify some SVA bitcode before it is passed to the `sva.translate()` intrinsic to trick the kernel into adding new, arbitrary code into the kernel.

To prevent such an attack, our design provides the `sva.disable.privcode()` instruction, which turns off code generation for the kernel. This will allow the kernel to dynamically load bitcode files for drivers and extend its native code section during boot but prevent further driver loading after boot. A kernel that loads all of its drivers during boot would use this instruction immediately before executing the first user process to limit the time at which it would be vulnerable to virtual code injection attacks. (Note that the OS feature to hot-swap devices that require loading new device drivers might be precluded by this design.)

6.4.10 Installing Interrupt and System Call Handlers

Operating systems designate functions that should be called when interrupts, traps, and system calls occur. Like SVA-M in Chapter 5, KCoFI provides instructions that allow the OS kernel to specify a function to handle a given interrupt, trap, or system call. These instructions first check that the specified address is within the kernel's virtual address space and has a CFI label. If the function address passes these checks, the instruction records the function address in a table that maps interrupt vector/system call numbers to interrupt/system call handling functions.

The hardware's interrupt vector table resides in KCoFI memory and directs interrupts into KCoFI's own interrupt and system call handling code. This code saves the interrupted program state as described in Section 6.4.6 and then passes control to the function that the kernel designated.

6.5 Formal Model and Proofs

In order to demonstrate that key features of our design are correct, we built a model of the KCoFI virtual machine in the Coq proof assistant [143] and provide a proof that our design enforces control-flow integrity. The model and proofs, given in Appendix B, comprise 5,695 non-comment lines of Coq code. Our proofs are checked mechanically by Coq.

As we are primarily interested in showing that our design in Section 6.4 is correct, we model a simplified version of the KCoFI VM. While our model is simpler than and not proven sound with respect to the full implementation, it models key features for which formal reasoning about control-flow integrity has not

Instructions	::=	loadi n
		| load n
		| store n
		| add n
		| sub n
		| map n *tlb*
		| jmp
		| jeq n
		| jneg n
		| trap
		| iret
		| svaSwap
		| svaRegisterTrap
		| svaInitIContext f
		| svaSaveIContext
		| svaLoadIContext
		| svaPushFunction n

Figure 6.4: Instructions in KCoFI Model. Most instructions take the single register, $\mathcal{R}$, as an implicit operand.

previously been done; these features include virtual to physical address translation, trap entry and return, context switching, and signal handler dispatch.

Our formal model and proofs provide several benefits. First, they provide a concise and formal description of the run-time checks that KCoFI must perform to enforce control-flow integrity. Second, the proofs help demonstrate that key features of our design are correct. Third, the process of proving the control-flow integrity theorems helped ensure that our formal specification of the run-time checks are correct: during the process of proving our control-flow integrity theorems, we realized that we had incorrectly specified how some of the run-time checks work when we determined why we couldn't prove one of the theorems. We fixed the errors and successfully completed the proof. The process of proving our control-flow integrity theorems helped reveal the error.

In this section, we describe our model of the KCoFI virtual machine, our formal definition of control-flow integrity for operating system code, and our control-flow integrity proofs.

6.5.1 KCoFI Virtual Machine Model

Our machine model is a simplified version of the KCoFI virtual machine with the instruction set shown in Figure 6.4. To simplify the language and its semantics, we opted to use a simple assembly language instruction set for basic computation instead of the SSA-based SVA instruction set. Operations such as context switching and MMU configuration are performed using instructions similar to those described in

Section 6.4. Our model does not include all the KCoFI features and does not model user-space code. However, it does include an MMU, traps, context switching, and the ability to modify and restore Interrupt Contexts as described in Section 6.4.6 (which is used to support signal handler dispatch).

The physical hardware is modeled as a tuple called the *configuration* that represents the current machine state. The configuration contains:

- the value of a single hardware register $\mathcal{R}$
- a program counter $\mathcal{PC}$
- a memory (or store) σ that maps physical addresses to values
- a software TLB μ that maps virtual addresses to TLB entries. A TLB entry is a tuple containing a physical address and three booleans that represent read, write, and execute permission to the physical address. The function ρ returns the physical address within a TLB entry while the functions RD(), WR(), and EX() return true if the TLB entry permits read, write, and execute access, respectively. Unlike real hardware, our model's MMU maps virtual to physical addresses at byte granularity.
- a set of virtual addresses $\mathcal{CFG}$ to which branches and traps may transfer control flow. All new threads must begin execution at a virtual address within $\mathcal{CFG}$.
- a pair (cs, ce) marking the first and last physical address of the kernel's code segment
- a current thread identifier $\mathcal{T}$
- a new thread identifier $\mathcal{NT}$
- a partial function τ that maps a thread identifier to a thread. A thread is a tuple $(v, pc, istack, sistack)$ in which v is a boolean that indicates whether a thread can be context switched on to the CPU and pc is the program counter at which execution should resume when the thread is loaded on to the CPU. The $istack$ is the Interrupt Context stack in Figure 6.3 used when traps and returns from traps occur. The $sistack$ is the Saved Interrupt Stack in Figure 6.3 and stores Interrupt Contexts that are saved by svaSaveIContext.
- a virtual address $\mathcal{TH}$ that is the address of the trap handler function

Function	Description
valid	$(v, pc, istack, sistack) \rightarrow v$
swapOn	$(v, n, istack, sistack) \times pc \rightarrow (true, pc, istack, sistack)$
swapOff	$(v, pc, istack, sistack) \rightarrow (false, 0, istack, sistack)$
ipush	$(v, pc, istack, sistack) \times ic \rightarrow (v, pc, ic :: istack, sistack)$
ipop	$(v, pc, ic :: istack, sistack) \rightarrow (v, pc, istack, sistack)$
itop	$(v, pc, ic :: istack, sistack) \rightarrow ic$
getIC	$(v, pc, istack, sistack) \rightarrow istack$
getSIC	$(v, pc, istack, sistack) \rightarrow sistack$
saveIC	$(v, pc, ic :: istack, sistack) \rightarrow (v, pc, ic :: istack, ic :: sistack)$
loadIC	$(v, pc, ic_1 :: istack, ic_2 :: sistack) \rightarrow (v, pc, ic_2 :: istack, sistack)$
length	$ic_1 :: istack \rightarrow 1 + \text{length}(istack)$ $nil \rightarrow 0$
defTH	$(v, pc, istack, sistack) \bigtimes \tau \rightarrow ((v, pc, istack, sistack) \in \tau)$
getIPC	$(\mathcal{R}, pc) \rightarrow pc$
getIReg	$(\mathcal{R}, pc) \rightarrow \mathcal{R}$

Table 6.5: Summary of Formal Model Support Functions

Since the configuration is large, we will replace one or more elements with an ellipsis (i.e., ...) as necessary to keep the text concise.

An Interrupt Context is a tuple that represents a subset of the configuration. It contains a copy of the machine's single register and the program counter. Interrupt Contexts are stored in stacks with standard push/pop semantics. The special value *nil* represents an empty stack; attempting to pop or read the top value of an empty stack results in an Interrupt Context with zero values.

There are several support functions, summarized in Table 6.5, that help make our semantics easier to read. The *valid* function takes a thread $\mathcal{T}$ and returns the value of its boolean flag. The *swapOn* function takes a thread $\mathcal{T}$ and an integer and returns an identical thread with its boolean flag set to true and its program counter set to the specified integer. Conversely, the *swapOff* function takes a thread and returns a thread that is identical except for its boolean being set to false and its program counter being set to zero. The *ipush* function takes a thread and an Interrupt Context and returns a thread with the Interrupt Context pushed on to the *istack* member of the thread tuple. The *ipop* function is similar but pops the top-most Interrupt Context off the thread's *istack*. The *getIC* function returns the Interrupt Context stack stored within a thread; the *getSIC* returns a thread's saved Interrupt Context stack. The *saveIC* function takes a thread and returns an identical thread in which the top-most element of the *istack* member is pushed on to the *sistack* member. The *loadIC* function pops the top-most Interrupt Context from the *sistack* member and uses that value to replace the top-most member of the *istack* member. The *itop* function takes a thread and returns the top-most Interrupt Context on its *istack*. The *length* function returns the number

of Interrupt Contexts stored in a stack of Interrupt Contexts. The *defTH* function returns true if a thread ID is defined within the thread function. Finally, the *getIPC* and *getIReg* functions take an Interrupt Context and return the program counter and register value stored within the Interrupt Context, respectively.

One feature of our configuration is that the KCoFI VM's internal data structures are not stored in memory; they are instead part of the configuration and can therefore not be modified by the `store` instruction. An advantage of this approach is that our proofs demonstrate that CFI is enforced regardless of the mechanism employed to protect these data structures (i.e., it shows that if these data structures are protected, then the proof holds). The disadvantage of this approach is that it does not prove that our sandboxing instrumentation on stores is designed correctly. However, given the simplicity of our instrumentation, we believe that having a simpler, more general proof is the better tradeoff.

6.5.2 Instruction Set and Semantics

The instruction set is shown in Figure 6.4. The `loadi` instruction loads the constant specified as its argument into the register; the `load` instruction loads the value in the virtual address given as its argument into the register. The `store` instruction stores the value in the register to the specified virtual memory address. The `add` (`sub`) instruction adds (subtracts) a constant with the contents of the register and stores the result in the register. The `map` instruction modifies a virtual to physical address mapping in the software-managed TLB. The `jmp` instruction unconditionally transfers control to the address in the register while the `jeq` and `jneg` instructions transfer control to the specified address if the register is equal to zero or negative, respectively.

A subset of the KCoFI instructions are also included in the instruction set. Some of the instructions differ from their KCoFI implementations because our formal model does not have an implicit stack whereas the KCoFI instruction set does. Our model also has `trap` and `iret` instructions for generating traps and returning from trap handlers.

The semantic rules for each instruction are specified as a state transition system. The transition relation $c_1 \Rightarrow c_2$ denotes that the execution of an instruction can move the state of the system from configuration c_1 to configuration c_2. Figure 6.5 shows the semantic rules for each instruction. Each rule has a brief name describing its purpose, the conditions under which the rule can be used, and then the actual transition relation. Each rule essentially fetches an instruction at the address of the program counter, checks for safety conditions (given as part of the premise of the implication), and then generates a new state for the machine to reflect the behavior of the instruction.

All instructions require that the program counter point to a virtual address with execute permission. Loads and stores to memory require read or write access, respectively. The jump instructions always check that the destination is a valid target. The map instruction is allowed to change a virtual to physical page mapping if the virtual address given as an argument is not already mapped to a location within the code segment *and* it does not permit a new virtual address to map to an address within the code segment.

6.5.3 Foundational Control-Flow Integrity Theorems

We now outline our control-flow integrity proofs for this system. Our first two proofs ensure that each transition in the semantics (i.e., the execution of a *single* instruction) maintains control-flow integrity.

There are several invariants that must hold on a configuration if the transition relation is to maintain control-flow integrity. For example, the system must not start in a state with a writeable code segment. We therefore define five invariants that should hold over all configurations:

Invariant 1. *VC(c): For configuration c = (..., cs, ce, ...), 0 < cs ≤ ce.*

Invariant 2. *TNW(c): For configuration c = (μ, σ, Reg, ..., cs, ce, ...), ∀ n : cs ≤ ρ(μ(n)) ≤ ce, ¬ WR(μ(n))*

Invariant 3. *TMAP1(c): For configuration c = (μ, σ, ..., cs, ce, ...), ∀ n m : cs ≤ ρ(μ(n)) ≤ ce ∧ n ≠ m, ρ(μ(n)) ≠ ρ(μ(m))*

Invariant 4. *TH(c): For configuration c = (..., $\mathcal{TH}$), $\mathcal{TH}$ ∈ CFG*

Invariant 5. *THR(c): For configuration c = (μ, σ, ..., CFG, ..., τ, ...), ∀ (v, pc, istack, sistack) ∈ τ: pc ∈ CFG ∨ σ(ρ(μ (pc - 1))) = svaSwap*

Invariant 1 states that the start of the code segment must be non-zero and less than or equal to the end of the code segment. Invariant 2 asserts that there are no virtual-to-physical address mappings that permit the code segment to be written. Invariant 3 asserts that there is at most one virtual address that is mapped to each physical address within the code segment. Invariant 4 ensures that the system's trap handler is an address that can be targeted by a branch instruction.

Invariant 5 restricts the value of the program counter in saved thread structures. A newly created thread needs to have an address at which to start execution; Invariant 5 restricts this value to being an address within $\mathcal{CFG}$. A thread that has been swapped off the CPU should have a program counter that points to the address immediately following the `svaSwap` instruction. The second half of the disjunction in Invariant 5 permits this.

LoadImm: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = loadi n $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, n, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

Load: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = load n $\wedge$ RD(μ(n)) $\wedge$ σ(ρ(μ(n))) = val v $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, v, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

Store: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = store n $\wedge$ WR(μ(n)) $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ[ρ(μ(n)) $\leftarrow$ (val $\mathcal{R}$)], $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

Add: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = add n $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$ + n, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

Sub: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = sub n $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$ - n, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

Jump: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = jmp $\wedge$ $\mathcal{R}$ $\in$ CFG $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, $\mathcal{R}$, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

JumpEq1: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = jeq v $\wedge$ v $\in$ CFG $\rightarrow$
(μ, σ, 0, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, 0, v, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

JumpEq2: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = jeq v $\wedge$ $\mathcal{R}$ $\neq$ 0 $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

JumpNeg1: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = jneg v $\wedge$ v $\in$ CFG $\wedge$ $\mathcal{R}$ < 0 $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, v, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

JumpNeg2: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = jneg v $\wedge$ $\mathcal{R}$ $\geq$ 0 $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

Map: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = map v tlb $\wedge$ $\neg$ (cs $\leq$ ρ(tlb) $\leq$ ce) $\wedge$ $\neg$ (cs $\leq$ ρ(μ(v)) $\leq$ ce) $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ[v $\leftarrow$ tlb], σ, $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$)

Swap: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = svaSwap $\wedge$ valid(τ($\mathcal{R}$)) $\wedge$ defTH($\mathcal{R}$,τ) $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{T}$, getPC(τ($\mathcal{R}$)), CFG, cs, ce, $\mathcal{R}$, $\mathcal{NT}$, τ[$\mathcal{T}$ $\leftarrow$ swapOn(τ($\mathcal{T}$), PC + 1)]
[$\mathcal{R}$ $\leftarrow$ swapOff(τ($\mathcal{R}$))], $\mathcal{TH}$)

Trap: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = trap $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, $\mathcal{TH}$, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ[$\mathcal{T}$ $\leftarrow$ ipush(τ($\mathcal{T}$),($\mathcal{R}$,PC + 1))], $\mathcal{TH}$)

IRet: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = iret $\wedge$ 0 < length (getIC(τ($\mathcal{T}$))) $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, getIReg(itop(τ($\mathcal{T}$))), getIPC(itop(τ($\mathcal{T}$))), CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ[$\mathcal{T}$ $\leftarrow$
ipop(τ($\mathcal{T}$))], $\mathcal{TH}$)

RegisterTrap: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = svaRegisterTrap $\wedge$ $\mathcal{R}$ $\in$ CFG $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{R}$)

InitIContext: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = svaInitIContext f $\wedge$ f $\in$ CFG $\wedge$ defTH($\mathcal{NT}$ + 1, τ) $\wedge$ 0 < length(getIC(τ($\mathcal{T}$))) $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{NT}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$ + 1, τ[$\mathcal{NT}$ $\leftarrow$ (true, f, itop(τ($\mathcal{T}$)) :: nil, nil)], $\mathcal{TH}$)

SaveIContext: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = svaSaveIContext $\wedge$ 0 < length (getIC(τ($\mathcal{T}$))) $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ[$\mathcal{T}$ $\leftarrow$ saveIC(τ($\mathcal{T}$))], $\mathcal{TH}$)

LoadIContext: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = svaLoadIContext $\wedge$ 0 < length (getSIC(τ($\mathcal{T}$))) $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ[$\mathcal{T}$ $\leftarrow$ loadIC(τ($\mathcal{T}$))], $\mathcal{TH}$)

PushIContext: EX(μ(PC)) $\wedge$ σ(ρ(μ(PC))) = svaPushFunction a $\rightarrow$
(μ, σ, $\mathcal{R}$, PC, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ, $\mathcal{TH}$) $\Rightarrow$ (μ, σ, $\mathcal{R}$, PC + 1, CFG, cs, ce, $\mathcal{T}$, $\mathcal{NT}$, τ[$\mathcal{T}$ $\leftarrow$ ipush(τ($\mathcal{T}$), (a, $\mathcal{R}$))], $\mathcal{TH}$)

Figure 6.5: KCoFI Instruction Semantics

Control-flow integrity in our system covers two key properties. First, each instruction should transfer control to one of four locations: the virtual address of the subsequent instruction, a valid target within the pre-computed control-flow graph, an instruction following an `svaSwap` instruction, or the program counter value stored in the top-most interrupt context of the current thread (which cannot be corrupted since it does not reside in memory). Theorem 1 states this more formally:

Theorem 1. $\forall\ c_1 = (\mu_1, \sigma_1, ..., PC_1, ..., CFG, ..., \mathcal{T}_1, ..., \tau_1),\ c_2 = (\mu_2, \sigma_2, ..., PC_2, ..., CFG, ..., \mathcal{T}_2, ..., \tau_2) : (c_1 \Rightarrow c_2) \wedge TH(c_1) \wedge THR(c_1) \rightarrow PC_2 = PC_1 + 1 \vee PC_2 \in CFG \vee \sigma_2(\rho(\mu_2(PC_2 - 1))) = svaSwap \vee PC_2 = getIPC(itop(\tau_1(\mathcal{T}_1)))$

Second, we want to ensure that the instruction stream read from the code segment does not change due to writes by the store instruction or by reconfiguration of the MMU. In other words, we want to ensure that reading from a virtual address that maps into the code segment reads the same value after executing an instruction as it held before execution of the instruction. Theorem 2 states this formally as:

Theorem 2. $\forall\ v,\ c_1 = (\mu_1, \sigma_1, ..., cs, ce, ...),\ c_2 = (\mu_2, \sigma_2, ...) : (c_1 \Rightarrow c_2) \wedge VC(c_1) \wedge TNW(c_1) \wedge TMAP1(c_1) \wedge cs \leq \rho(\mu_1(v)) \leq ce \rightarrow \sigma_1(\rho(\mu_1(v))) = \sigma_2(\rho(\mu_2(v)))$

We proved that both Theorem 1 and 2 hold true in the Coq proof assistant. Intuitively, Theorem 1 is true because all the jump instructions check the target address against the precomputed control-flow graph while the `svaSwap` instruction always saves and loads the correct program value from saved processor state. The intuition behind Theorem 2 is that the checks on the `map` instruction prevent a) addresses that are already mapped into the code segment from being remapped to different addresses, and b) new virtual-to-physical address mappings from making parts of the code segment writable.

6.5.4 Complete Control-Flow Integrity Theorems

While proving Theorems 1 and 2 shows that the restrictions on the instructions maintain control-flow integrity for single instruction executions, a full and complete control-flow integrity proof demonstrates that control-flow integrity is maintained on the transitive closure of the transition relation (in other words, if the system starts in a state satisfying the required invariants, then control-flow integrity is maintained after executing an arbitrary number of instructions). Such a proof requires that the transition relation preserve the invariants. Proofs that the invariants hold over the transition relation rely upon other additional invariants holding over the transition relation.

We have identified a set of invariants, in addition to those identified in Section 6.5.3, that can be used to show that control-flow integrity holds over the reflexive and transitive closure of the transition relation, denoted $\Rightarrow^*$. Using Coq, we have proven that combinations of these invariants hold over the transition relation and its reflexive, transitive closure. These additional invariants are:

Invariant 6. *CFGT(c): For configuration c = (μ, ..., CFG, cs, ce, ...), ∀ v: v ∈ CFG → cs ≤ ρ(μ(v)) ≤ ce.*

Invariant 7. *PCT(c): For configuration c = (μ, ..., PC, ..., cs, ce, ...), cs < ρ(μ(PC)) ≤ ce.*

Invariant 8. *tlText(c): For configuration c = (μ, ..., CFG, cs, ce, ..., τ, ...), ∀ (v, pc, istack, sistack) ∈ τ: cs ≤ ρ(μ(pc)) ≤ ce ∧ ((pc ∈ CFG) ∨ cs ≤ ρ(μ(pc - 1)) ≤ ce)*

Invariant 9. *textMappedLinear(c): For configuration c = (μ, ..., cs, ce, ...), ∀ v : (cs ≤ ρ(μ(v)) ≤ ce) → (cs ≤ ρ(μ(v + 1)) ≤ ce) ∨ σ(ρ(μ(v))) = jmp*

Invariant 10. *IC(c): For configuration c = (μ, ..., CFG, cs, ce, ..., τ, ...), ∀ (v, pc, istack, sistack) ∈ τ: ∀ (r, pc) ∈ istack: (cs ≤ ρ(μ(pc)) ≤ ce) ∧ (cs ≤ ρ(μ(pc-1)) ≤ ce)*

Invariant 11. *SIC(c): For configuration c = (μ, ..., CFG, cs, ce, ..., τ, ...), ∀ (v, pc, istack, sistack) ∈ τ: ∀ (r, pc) ∈ sistack: (cs ≤ ρ(μ(pc)) ≤ ce) ∧ (cs ≤ ρ(μ(pc-1)) ≤ ce)*

Invariant 12. *VTID(c): For configuration c = (..., ..., $\mathcal{T}$, $\mathcal{NT}$, τ, ...), defTH($\mathcal{T}$, τ) ∧ defTH($\mathcal{NT}$, τ)*

Invariants 6 and 7 state that the list of valid indirect branch targets and the machine's program counter are all virtual addresses that are mapped to the code segment. Invariant 8 states that all swapped-off threads also have program counters that are within the code segment; it also ensures that threads with program counters that are not within the valid list of indirect branch targets have their previous program counter within the code segment (i.e., the `svaSwap` instruction that swapped the thread off the CPU is also in the code segment).

Invariant 9 requires that, for each virtual address mapped to the text segment, the virtual address be a `jmp` instruction or the subsequent virtual address by mapped into the text segment also.

Invariants 10 and 11 state that the program counter values stored within the Interrupt Contexts within a thread are all virtual addresses that are within the code segment; like Invariant 8, they also require the previous virtual address to be in the code segment as well. Proving that these invariants hold over the step relation permits us to prove that the equivalents of Theorem 1 and Theorem 2 hold over the reflexive, transitive closure of the step relation. They are expressed as follows:

Theorem 3. $\forall\ c_1,\ c_2,\ c_3 = (\mu_3,\ \sigma_3,\ ...,\ PC_3,\ ...,\ CFG,\ ...,\ \mathcal{T}_3,\ ...,\ \tau_3),\ c_4 = (\mu_4,\ \sigma_4,\ ...,\ PC_4,\ ...,\ CFG,\ ...,\ \mathcal{T}_4,\ ...,\ \tau_4) :\ (c_1 \Rightarrow^* c_3) \wedge (c_3 \Rightarrow c_4) \wedge (c_4 \Rightarrow^* c_2) \wedge PCT(c_1) \wedge VTID(c_1) \wedge CFGT(c_1) \wedge textMappedLinear(c_1) \wedge IC(c_1) \wedge SIC(c_1) \wedge THR(c_1) \wedge TH(c_1) \wedge TMAP1(c_1) \wedge tlText(c_1) \rightarrow PC_4 = PC_3 + 1 \vee PC_4 \in CFG \vee \sigma_4(\rho(\mu_4(PC_4 - 1))) = svaSwap \vee PC_4 = getIPC(itop(\tau_3(\mathcal{T}_3)))$

Theorem 4. $\forall\ v,\ c_1 = (\mu_1,\ \sigma_1,\ ...,\ cs,\ ce,\ ...),\ c_2 = (\mu_2,\ \sigma_2,\ ...) :\ (c_1 \Rightarrow^* c_2) \wedge VC(c_1) \wedge TNW(c_1) \wedge TMAP1(c_1) \wedge cs \leq \rho(\mu_1(v)) \leq ce \rightarrow \sigma_1(\rho(\mu_1(v))) = \sigma_2(\rho(\mu_2(v)))$

Theorem 3 states that if the invariants hold on an initial configuration (i.e., c_1), and if the system can execute an arbitrary number of instructions to reach an end configuration (i.e., c_2), then every transition that occurs while moving from the first configuration to the end configuration does so while maintaining control-flow integrity. Intuitively, this is true because all of the invariants are held for each configuration visited between configurations c_1 and c_2, allowing Theorem 1 to be applied repeatedly along the chain of transitions.

Theorem 4 states that the values read from the code segment do not change regardless of the number of instructions executed between the two configurations. Like Theorem 3, it is true because the invariants needed by Theorem 2 hold for each configuration visited between the initial and final configuration, allowing Theorem 2 to be applied repeatedly.

6.6 Implementation

KCoFI uses the 64-bit version of SVA-OS described in Chapter 3. For our experiments, we use the FreeBSD 9.0 kernel that was ported to SVA.

We used the sloccount tool [151] from the FreeBSD ports tree to measure the size of our TCB. Excluding comments and white space, our system contains 4,585 source lines of code for the KCoFI run-time library linked into the kernel and an additional 994 source lines of code added to the LLVM compiler to implement the compiler instrumentation. In total, our TCB is 5,579 source lines of code.

6.6.1 Instrumentation

The CFI and store instrumentation is implemented in two separate LLVM passes. The CFI instrumentation pass is a version of the pass written by Zeng et. al. [160] that we updated to work on x86_64 code with LLVM 3.1. The store instrumentation pass is an LLVM IR level pass that instruments store and atomic instructions that modify memory; it also instruments calls to LLVM intrinsic functions such as `llvm.memcpy`.

We modified the Clang/LLVM 3.1 compiler to utilize these instrumentation passes when compiling kernel code. To avoid the need for whole-program analysis, we use a very conservative call graph: we use one label for all call sites (i.e., the targets of returns) and for the first address of every function. While conservative, this callgraph allows us to measure the performance overheads and should be sufficient for stopping advanced control-data attacks.

Unlike previous work [160, 159], we use the sequence `xchg %rcx, %rcx` ; `xchg %rdx, %rdx` to create a 32-bit label. We found that this sequence is both easy to implement (since they are NOPs, these instructions do not overwrite any register values) and much faster than a 64-bit version of the `prefetchnta` sequence used in previous work [160].

6.6.2 KCoFI Instruction Implementation

The KCoFI instructions described in Section 6.4 are implemented in a run-time library that is linked into the kernel at compile-time. The semantics for the instructions given in Section 6.5 assume that the KCoFI instructions execute atomically. For that reason, the run-time library implementations disable hardware interrupts when they start execution and re-enable them upon exit as the original SVA-M implementation in Chapter 5 did.

6.6.3 Interrupt Context Implementation

To place the Interrupt Context within the KCoFI VM internal memory, we use the Interrupt Stack Table (IST) feature of the x86_64 [4]. This feature causes the processor to change the stack pointer to a specific location on traps or interrupts regardless of whether a processor privilege mode change has occurred. The KCoFI VM uses this feature to force the processor to save state within KCoFI's internal memory before switching to the real kernel stack.

Unlike the version of SVA-M presented in Chapters 4 and 5, KCoFI saves all native processor state on every interrupt, trap, and system call. This includes the x86_64 general purpose registers, the x87 FPU registers, and the XMM and SSE registers. We believe an improved implementation can save the floating point and vector state lazily as the native FreeBSD 9.0 kernel does, but that improvement is left to future work.

6.6.4 Unimplemented Features

Our implementation does not yet include the protections needed for DMA. However, we believe that I/O MMU configuration is rare, and therefore, the extra protections for DMA should not add noticeable overhead. Our implementation also lacks the ability to translate SVA bitcode (or to look up cached translations for bitcode) at run-time. Instead, our current implementation translates all OS components to native code ahead-of-time, and runs only native-code applications.

For ease of implementation, we add the same CFI labels to both kernel code and the SVA-OS run-time library. While this deviates from the design, it does not change the performance overheads and makes the security results more conservative (no better and possibly worse).

6.7 Security Evaluation

We performed two empirical evaluations to measure the security of our approach. Since KCoFI does not permit memory to be both readable and executable, we focus on return-oriented programming attacks.

Our first evaluation examines how well KCoFI removes instructions from the set of instructions that could be used in a return-oriented programming attack (which can work with or without return instructions [35]). We compute a metric that summarizes this reduction called the average indirect target reduction (AIR) metric [162].

Since not all instructions are equally valuable to an attacker, we performed a second evaluation that finds instruction sequences (called *gadgets* [118]) that can be used in an ROP attack and determines whether they can still be used after KCoFI has applied its instrumentation.

6.7.1 Average Indirect Target Reduction

Return oriented programming attacks work because of the plethora of instructions available within a program. To get a sense of how many instructions we removed from an attacker's set of usable instructions, we used Zhang and Sekar's AIR metric [162]; this metric computes the average number of machine code instructions that are eliminated as possible targets of indirect control transfers. The AIR metric quantifies the reduction in possible attack opportunities in a way that is independent of the specific CFI method employed, the compiler used, and the architecture.

Equation 6.1 from Zhang and Sekar [162] provides the general form for computing the AIR metric for a program. n is the number of indirect branch instructions in the program, S is the total number of

instructions to which an indirect branch can direct control flow before instrumentation, and $|T_i|$ is the number of instructions to which indirect branch i can direct control flow after instrumentation:

$$\frac{1}{n}\sum_{j=1}^{n} 1 - \frac{|T_j|}{S} \tag{6.1}$$

Since all indirect branch instructions instrumented by KCoFI can direct control-flow to the same set of addresses, Equation 6.1 can be simplified into Equation 6.2 (with $|T|$ being the number of valid targets for each indirect branch):

$$1 - \frac{|T|}{S} \tag{6.2}$$

We measured the AIR metric for the KCoFI native code generated for the FreeBSD kernel. Our compiler identified 106,215 valid native code targets for indirect control flow transfers ($|T|$) out of 5,838,904 possible targets in the kernel's code segment (S) before instrumentation. The native code generated by KCoFI contains 21,635 indirect control flow transfers (n). The average reduction of targets (AIR metric) for these transfers is therefore 98.18%, implying that nearly all the possible indirect control transfer targets have been eliminated as feasible targets by KCoFI.

As a point of comparison, our AIR result is nearly as good as the average AIR metrics for several different CFI variants reported for the SPEC CPU 2006 benchmarks and the namd benchmark (which range between 96% to 99.1%) [162]. Since these numbers are obtained for very different workloads – SPEC and the FreeBSD kernel – the comparison is only intended to show that the results are roughly similar; the differences in the exact numbers are not meaningful.

6.7.2 ROP Gadgets

To measure the impact on return-oriented-programming opportunities more specifically, we used the open-source ROPGadget tool [121] version 4.0.4 to automatically find ROP gadgets in both the original FreeBSD kernel compiled with GCC and our identically configured KCoFI FreeBSD kernel. We ran the tool on both the kernel and drivers using the default command-line options.

ROPGadget found 48 gadgets in the original FreeBSD kernel and 21 gadgets in the KCoFI FreeBSD kernel. We manually analyzed the 21 gadgets found in the KCoFI FreeBSD kernel. None of the gadgets follow a valid control-flow integrity label. Therefore, none of these gadgets can be "jumped to" via an indirect control transfer in an ROP attack.

6.8 Performance Evaluation

We evaluated the performance impact of KCoFI on a Dell Precision T1650 workstation with an Intel® Core™ i7-3770 processor at 3.4 GHz with 8 MB of cache, 16 GB of RAM, an integrated PCIE Gigabit Ethernet card, a 7200 RPM 6 Gb/s SATA hard drive, and a Solid State Drive (SSD) used for the `/usr` partition. For experiments requiring a network client, we used an iMac with a 4-core hyper-threaded Intel® Core™ i7 processor at 2.6 GHz with 8 GB of RAM. Our network experiments used a dedicated Gigabit Ethernet network.

Since network applications make heavy use of operating system services, we measured the performance of the `thttpd` web server and the remote secure login `sshd` server. These experiments also allow us to compare the performance of KCoFI to the original SVA-M system in Chapter 5 which enforces a more sophisticated memory safety policy that implies control-flow integrity.

To measure file system performance, we used the Postmark benchmark [116]. We used the LMBench microbenchmarks [100] to measure the performance of individual system calls.

For each experiment, we booted the Dell machine into single-user mode to avoid having other system processes affect the performance of the system. Our baseline is a native FreeBSD kernel compiled with the LLVM 3.1 compiler, with the same compiler options and the same kernel options as the KCoFI FreeBSD kernel.

6.8.1 Web Server Performance

We used a statically linked version of the `thttpd` web server [115] to measure how much the KCoFI run-time checks reduce the server's bandwidth. To measure bandwidth, we used ApacheBench [1].

For the experiments, we transferred files between 1 KB and 2 MB in size. Using larger file sizes is not useful because the network saturates at about 512KB file sizes. This range of sizes also subsumes the range used in the original SVA-M experiments in Chapter 5. We generated each file by collecting random data from the `/dev/random` device; this ensures that the file system cannot optimize away disk reads due to the file having blocks containing all zeros. We configured each ApacheBench client to make 32 simultaneous connections and to perform 10,000 requests for the file; we ran four such ApacheBench processes in parallel for each run of the experiment to simulate multiple clients. We ran each experiment 20 times.

Figure 6.6 shows the mean performance of transferring a file of each size. The average bandwidth reduction across all file sizes is essentially zero. This is far better performance than the SVA-M system which incurs about a 25% reduction in bandwidth due to its memory safety checks (see Chapter 5).

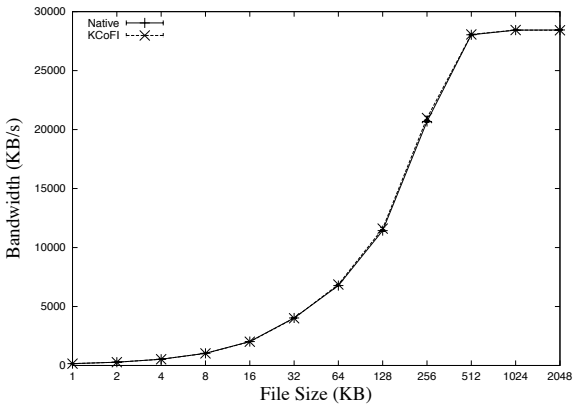


Figure 6.6: ApacheBench Average Bandwidth with Standard Deviation Bars

6.8.2 Secure Shell Server Performance

In addition to a web server, we also measured the bandwidth of transferring files using the OpenSSH Secure Shell server [117]. We ran the OpenSSH server on our test machine and used the Mac OS X OpenSSH scp client (based on OpenSSH 5.2p1) to measure the number of bytes received per second when transferring the file. We repeated each experiment 20 times.

Figure 6.7 plots the mean bandwidth for the baseline system and KCoFI with standard deviation error bars (the standard deviations are too small to be discernible in the diagram). On average, the bandwidth reduction was 13% with a worst case reduction of 27%. Transferring files between 1 KB and 8 KB showed the most overhead at 27%. Transferring files that are 1 MB or smaller showed an average overhead of 23%; the average is 2% for files of larger size, indicating that the network becomes the bottleneck for larger file transfers.

The original SVA-M system only measured SSH bandwidth for files that were 8 MB or larger (see Chapter 5); this is beyond the point at which the network hardware becomes the bottleneck. This comparison, therefore, is inconclusive: it does not show any difference between the two systems, but it does not include cases where overheads might be expected.

6.8.3 Microbenchmarks

In order to understand how our system affects the performance of core operating system services, we used LMBench [100] to measure the latency of various system calls. (We present these before discussing Postmark because the latter is largely explained by the LMBench measurements.) Some test programs can be configured

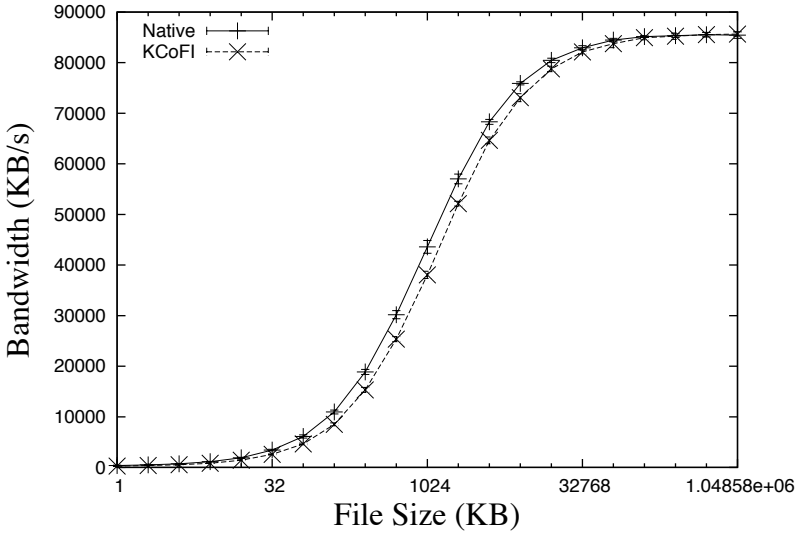


Figure 6.7: SSHD Average Transfer Rate with Standard Deviation Bars

to run the test for a specified number of iterations; those were configured to use 1,000 iterations. We ran each benchmark 10 times. We configured file I/O benchmarks to use files on the SSD. This ensures that we're measuring the highest relative latency that KCoFI can add by using the fastest disk hardware available.

Test	**Native**	**KCoFI**	**Overhead**	**SVA Overhead [49]**
null syscall	0.091	0.22	2.50x	2.31x
open/close	2.01	4.96	2.47x	11.0x
mmap	7.11	23.4	3.30x	-
page fault	31.6	35.2	1.11x	-
signal handler install	0.168	0.36	2.13x	5.74x
signal handler delivery	1.27	1.17	0.92x	5.34x
fork + exit	62.9	222	3.50x	-
fork + exec	101	318	3.10x	-
select	3.05	4.76	1.60x	8.81x
pipe latency	1.94	4.01	2.10x	13.10x

Table 6.6: LMBench Results. Time in Microseconds.

As Tables 6.6 and 6.7 show, our system can add considerable overhead to individual operations. Most of the operations we tested showed overhead between 2x and 3.5x. The KCoFI file creation overheads are uniformly about 2.3-2.5x for all file sizes tested by LMBench. Although these overheads are fairly high, most applications only experience these overheads during kernel CPU execution, which explains why the impact on performance observed for `thttpd` and `sshd` is far lower.

File Size	Native	KCoFI	Overhead
0 KB	155771	68415	2.28x
1 KB	97943	39615	2.47x
4 KB	97192	39135	2.48x
10 KB	85600	35982	2.38x

Table 6.7: LMBench: Files Creations Per Second

We also compared these results to similar benchmarks from the full memory-safety version of SVA-M (Chapter 5), shown in the last column of Table 6.6. The missing numbers for SVA-M are because some kernel operations were not tested in the SVA-M experiments. On these microbenchmarks, KCoFI clearly performs much better than SVA-M, as much as 5x in some cases. Again, the SVA-M experiments used a different kernel and so it is not meaningful to compare the detailed differences in the numbers, but the magnitudes of the differences clearly highlight the performance benefit of using CFI instead of full memory safety.

6.8.4 Postmark Performance

To further examine file system performance, we ran Postmark [116] which mimics a mail server's file system behavior.

Native (s)	Native StdDev	KCoFI (s)	KCoFI StdDev	Overhead
12.7	0.48	24.8	0.40	1.96x

Table 6.8: Postmark Results

We configured Postmark to use 500 base files with sizes ranging from 500 bytes to 9.77 KB with 512 byte block sizes. The read/append and create/delete biases were set to 5, and we configured Postmark to use buffered file I/O. We ran Postmark on the SSD since it has lower latency and less variability in its latency than the hard disk. Each run of the experiment performed 500,000 transactions. We ran the experiment 20 times on both the native FreeBSD kernel and the KCoFI system. Table 6.8 shows the average results.

As Table 6.8 shows, the Postmark overheads are close to the LMBench file creation overheads.

6.9 Related Work

Abadi et. al. [10] introduced the definition of control-flow integrity and proved that their approach enforced context-insensitive control-flow integrity. Our proof for the KCoFI design uses a similar approach but also demonstrates how control-flow integrity is preserved during OS operations that can have complex,

often unanalyzable, impact on control flow, including context switching, MMU configuration, signal handler dispatch, and interrupts.

Zhang and Sekar's BinCFI [162] and Zhang et. al.'s CCFIR [161] transform binary programs to enforce CFI. Similarly, Strato [159] modifies the LLVM compiler to instrument code with CFI checks similar to those used by KCoFI. None of these techniques can protect against ret2usr attacks since they find the targets of control-flow transfers via static analysis. KCoFI does not verify that its instrumentation is correct like Strato does [159]. However, KCoFI can incorporate Strato's instrumentation verification techniques.

RockSalt [105] is a verified verifier for Google's Native Client [158]; the verifier ensures that Native Client x86 machine code enforces sandboxing and control-flow integrity properties and has been proven correct via formal proof. Native Client's CFI policy [158] only requires that the x86 segment registers be left unmodified and that branches jump to aligned instructions; its CFI policy is therefore less restrictive than KCoFI's policy, and it does not permit code to perform the myriad of operations that an OS kernel must be able to perform.

ARMor [163] rewrites ARM applications to enforce control-flow integrity and software fault isolation; it uses an automated proof assistant and formal semantics of the ARM processor to verify that the transformed application enforces control-flow integrity. ARMor's formalism can reason about control transfers for which the targets can be computed ahead of time; however, it does not appear to reason about returns from interrupts (for which a target cannot be determined statically).

The Secure Virtual Architecture (Chapters 4 and 5) provides strong control-flow integrity guarantees. However, it also enforces very strong memory safety properties and sound points-to analysis; this required the use of whole-program pointer analysis [88] which is challenging to implement. SafeDrive [164] also enforces memory safety on commodity OS kernel code. However, SafeDrive requires the programmer to insert annotations indicating where memory object bounds information can be located. These annotations must be updated as the code is modified or extended.

HyperSafe [149] enforces control-flow integrity on a hypervisor. Like SVA-M (Chapter 5), HyperSafe vets MMU translations to protect the code segment and interrupt handler tables; it also introduces a new method of performing indirect function call checks. HyperSafe, however, does not protect the return address in interrupted program state, so it does not fully implement CFI guarantees and does not prevent ret2usr attacks. Furthermore, HyperSafe only protects a hypervisor, which lacks features such as signal handler delivery; KCoFI protects an entire commodity operating system kernel.

kGuard [82] prevents ret2usr attacks by instrumenting kernel code to ensure that indirect control flow transfers move control flow to a kernel virtual address; it also uses diversification to prevent attacks from

bypassing its protection and to frustrate control-flow hijack attacks. KCoFI uses similar bit-masking as kGuard to prevent user-space native code from forging kernel CFI labels. kGuard also uses diversification to prevent their instrumentation from being bypassed, which provides probabilistic protection against ROP attacks (with relatively low overhead), whereas KCoFI provides a CFI guaranty and ensures that the kernel's code segment is not modified.

Giuffrida et. al. [69] built a system that uses fine-grained randomization of the kernel code to protect against memory safety errors. Their system's security guarantees are probabilistic while our system's security guarantees are always guaranteed. Additionally, their prototype has only been applied to Minix while ours has been applied to a heavily used, real-world operating system (FreeBSD).

SecVisor [126] prevents unauthorized code from executing in kernel space but does not protect loaded code once it is executing. Returnless kernels [90] modify the compiler used to compile the OS kernel so that the kernel's binary code does not contain return instructions. Such kernels may still have gadgets that do not utilize return instructions [35].

The seL4 [83] microkernel is written in a subset of C and both the design and implementation are proven functionally correct, using an automated proof assistant. The proof ensures that the code does not have memory safety errors that alter functionality [83]. While seL4 provides stronger security guarantees than KCoFI, it only provides them to the microkernel while KCoFI provides its guarantees to a commodity OS kernel of any size. Changes to the seL4 code must be correct and require manual updates to the correctness proof [83] while KCoFI can automatically reapply instrumentation after kernel changes to protect buggy OS kernel code.

Several operating systems provide control-flow integrity by virtue of being written in a memory-safe programming language [28, 122, 74, 70, 156]. Verve [156], the most recent, is a kernel written in a C#-like language that sits upon a hardware abstraction layer that has been verified to maintain the heap properties needed for memory safety. While KCoFI can enforce control-flow integrity, its implementation is not verified like Verve's hardware abstraction layer.

6.10 Summary

In this chapter, we presented KCoFI: a system which provides comprehensive control-flow integrity to commodity operating systems. We have shown that KCoFI provides protection to OS kernel code similar to that found for user-space code with better overheads than previously developed techniques for commodity OS kernels. Essentially, KCoFI uses traditional label-based protection for programmed indirect jumps but

adds a thin run-time layer linked into the OS that protects key OS kernel data like interrupted program state and monitors all low-level state manipulations performed by the OS. We have provided a proof that KCoFI's design correctly enforces CFI, adding confidence in the correctness of our system.

Chapter 7

Protecting Applications from Compromised Operating Systems

7.1 Introduction

Applications that process sensitive data on modern commodity systems are vulnerable to compromises of the underlying system software. The applications themselves can be carefully designed and validated to be impervious to attack. However, all major commodity operating systems use large monolithic kernels that have complete access to and control over all system resources [119, 130, 30, 99]. These operating systems are prone to attack [94] with large attack surfaces, including large numbers of trusted device drivers developed by numerous hardware vendors and numerous privileged applications that are as dangerous as the operating system itself. A compromise of any of these components or of the kernel gives the attacker complete access to all data belonging to the application, whether in memory or offline. Developers of secure applications running on such a system generally have no control over any of these components and must trust them implicitly for their own security.

Several previous projects [38, 157, 75, 96] have created systems that protect applications from a compromised, or even a hostile, operating system. These systems have all used hardware page protection through a trusted hypervisor to achieve control over the operating system capabilities. They rely on a technique called *shadowing* or *cloaking* that automatically encrypts (i.e., shadows or cloaks) and hashes any application page that is accessed by the operating system, and then decrypts it and checks the hash when it is next accessed by the application. System call arguments must be copied between secure memory and memory the OS is allowed to examine. While these solutions are effective, they have several drawbacks. First, they rely upon encrypting any of an application's memory that is accessed by the OS; an application cannot improve performance by protecting only a selected subset of data, or requesting only integrity checks on data but not confidentiality (i.e., only using hashing and not encryption). Second, they assume the OS runs as a guest on a standard hypervisor, which may not be attractive in certain settings, such as energy-constrained mobile devices. Third, they require that all system call arguments must always be copied, even if the data being

transferred is not security-sensitive, which is the common case in many applications. Fourth, these solutions do not provide any security benefits to the kernel itself; for example, control flow integrity or memory safety for the operating system kernel cannot reuse the mechanisms developed for shadowing.

We propose a new approach we call *ghosting* that addresses all these limitations. Our system, *Virtual Ghost*, is the first to enforce application security from a hostile OS using compiler instrumentation of operating system code; this is used to create secure memory called *ghost memory*, which cannot be read or modified *at all* by the operating system (in contrast, previous systems like Overshadow [38] and InkTag [75] do not prevent such writes and only guarantee that the tampering will be detected before use by the application). Virtual Ghost introduces a thin hardware abstraction layer that provides a set of operations the kernel must use to manipulate hardware, and the secure application can use to obtain essential trusted services for ghost memory management, encryption, signing, and key management. Although the positioning of, and some of the mechanisms in, this layer are similar to a hypervisor, Virtual Ghost is unique because (a) unlike a traditional hypervisor, there is *no* software that runs at a higher privilege level than the kernel – in particular, the hardware abstraction layer runs at the same privilege level; (b) Virtual Ghost uses (simple, reliable) compiler techniques rather than hardware page protection to secure its own code and data; and (c) Virtual Ghost completely denies OS accesses to secure memory pages, not just encrypting and signing the pages to detect OS tampering.

Moreover, the compiler instrumentation in Virtual Ghost inherently provides strong protection against external exploits of the OS. First, traditional exploits, such as those that inject binary code, are not even expressible: all OS code *must* first go through SVA bitcode form and be translated to native code by the Virtual Ghost compiler. Second, attacks that leverage existing native code, like return-oriented programming (ROP) [118], require control-flow hijacking, which Virtual Ghost explicitly prevents as well. In particular, Virtual Ghost enforces Control Flow Integrity (CFI) [10] on kernel code in order to ensure that the compiler instrumentation of kernel code is not bypassed. CFI automatically defeats control-flow hijacking attacks, including the latter class of external exploits. Together, these protections provide an additional layer of defense for secure applications on potentially buggy (but non-hostile) operating systems.

Another set of differences from previous work is in the programming model. First, applications can use ghost memory selectively for all, some, or none of their data. When using it for all their data, the secure features can be obtained transparently via a modified language library (e.g., libc for C applications), similar to previous work [75]. Second, applications can pass non-ghost memory to system calls without the performance overheads of data copying. Third, when sending sensitive data through the operating system

(e.g., for I/O), ghost applications can choose which encryption and/or cryptographic signing algorithms to use to obtain desired performance/security tradeoffs, whereas previous systems generally baked this choice into the system design. Finally, a useful point of similarity with previous work is that the usability features provided by previous systems (e.g., secure file system services in InkTag) are orthogonal to the design choices in Virtual Ghost and can be directly incorporated.

We developed a prototype of Virtual Ghost and ported the FreeBSD 9.0 kernel to it using the SVA infrastructure described in Chapter 3. SVA provides the compiler instrumentation capabilities that Virtual Ghost needs. SVA also provides the ability to identify and control interactions between the system software and applications; this feature is crucial for reigning in OS kernel behavior so that the OS kernel cannot attack applications.

To evaluate Virtual Ghost's effectiveness, we ported three important applications from the OpenSSH application suite to Virtual Ghost, using ghost memory for the heap: `ssh`, `ssh-keygen`, and `ssh-agent`. These three can exchange data securely by sharing a common application key, which they use to encrypt the private authentication keys used by the OpenSSH protocols.

Since exploiting a kernel that runs on Virtual Ghost via an external attack is difficult, we evaluated the effectiveness of Virtual Ghost by adding a malicious module to the FreeBSD kernel, replacing the `read` system call handler. This module attempted to perform two different attacks on `ssh-agent`, including a sophisticated one that tries to alter application control flow via signal handler dispatch. When running without Virtual Ghost, both exploits successfully steal the desired data from `ssh-agent`. Under Virtual Ghost, both exploits fail and `ssh-agent` continues execution unaffected.

Our performance results show that Virtual Ghost outperforms InkTag [75] on five out of seven of the LMBench microbenchmarks, with improvements between 1.3x and 14.3x. The overheads for applications that perform moderate amounts of I/O (thttpd and sshd) is negligible, but the overhead for a completely I/O-dominated application (postmark) was high. We are investigating ways to reduce the overhead of postmark.

The rest of the chapter is organized as follows. Section 7.2 describes the attack model that we assume in this work. Section 7.3 gives an overview of our system, including the programmer's view when using it and the security guarantees the system provides. Section 7.4 describes our design in detail, and Section 7.5 describes the implementation of our prototype. Section 7.6 describes our modifications to secure OpenSSH applications with our system. Sections 7.7 and 7.8 evaluate the security and performance of our system. Section 7.9 describes related work, and Section 7.10 concludes.

7.2 System Software Attacks

In this section, we briefly describe our threat model and then describe the attack vectors that a malicious operating system might pursue within this threat model.

7.2.1 Threat Model

We assume that a user-space (i.e., unprivileged) application wishes to execute securely and perform standard I/O operations, but without trusting the underlying operating system kernel or storage and networking devices. Our goal is to preserve the application's integrity and confidentiality. Availability is outside the scope of the current work; we discuss the consequences of this assumption further, below. We also do not protect against side-channel attacks or keyboard and display attacks such as stealing data via keyboard loggers or from graphics memory; previous software solutions such as Overshadow [38] and InkTag [75] do not protect against these attacks either, whereas hardware solutions such as ARM's TrustZone [24] and Intel's SGX [98] do.

We assume that the OS, including the kernel and all device drivers, is malicious, i.e., may execute arbitrary hostile code with maximum privilege on the underlying hardware. We *do not assume that a software layer exists that has higher privilege than the OS*. Instead, we assume that the OS source code is *ported* to a trusted run-time library of low-level functions that serve as the interface to hardware (acting as a hardware abstraction layer) and supervise all kernel-hardware interactions. The OS is then compiled using a modified compiler that instruments the code to enforce desired security properties, described in later sections.[1] We assume that the OS can load and unload arbitrary (untrusted) OS modules dynamically, but these modules must also be compiled by the instrumenting compiler. Moreover, we assume that the OS has full read and write access to persistent storage, e.g., hard disk or flash memory.

We do *not* prevent attacks against the application itself. In practice, we expect that a secure application will be carefully designed and tested to achieve high confidence in its own security. Moreover, in practice, we expect that a secure application (or the secure subsets of it) will be much smaller than the size of a commodity OS, together with its drivers and associated services, which typically run into many millions of lines of code. For all these reasons, the developers and users of a secure application will not have the same level of confidence in a commodity OS as they would in the application itself.

[1] It is reasonable to expect the OS to be ported and recompiled because, in all the usage scenarios described in Section 7.1, we expect OS developers to make it an explicit design goal to take the OS out of the trusted computing base for secure applications.

Application availability is outside the scope of the current work. The consequence, however, is only that an attacker could deny a secure application from making forward progress (which would likely be detected quickly by users or system administrators); she could not steal or corrupt data produced by the application, even by subverting the OS in arbitrary ways.

7.2.2 Attack Vectors

Within the threat model described in Section 7.2.1, there are several attack vectors that malicious system software can take to violate an application's confidentiality or integrity. We describe the general idea behind each attack vector and provide concrete example attacks.

Data Access in Memory

The system software can attempt to access data residing in application memory. Examples include:

- The system software can attempt to read and/or write application memory directly via load and store instructions to application virtual addresses.
- Alternatively, the OS may attempt to use the MMU to either map the physical memory containing the data into virtual addresses which it can access (reading), or it may map physical pages that it has already modified into the virtual addresses that it cannot read or write directly (writing).
- The system software can direct an I/O device to use DMA to copy data to or from memory that the system software cannot read or write directly and memory that the system software can access directly.

Data Access through I/O

A malicious OS can attempt to access data residing on I/O devices or being transferred during I/O operations. Examples include:

- The OS can read or tamper with data directly from any file system used by the application.
- The OS can read or tamper with data being transferred via system calls to or from external devices, including persistent storage or networks.
- The OS can map unexpected blocks of data from I/O devices into memory on an `mmap` system call, effectively substituting arbitrary data for data expected by the application.

Code Modification Attacks

A malicious OS may attempt to change the application code that operates upon application data, in multiple ways, so that the malicious code would execute with the full memory access and I/O privileges of the application. Some examples:

- The OS can attempt to modify the application's native code in memory directly.
- The OS could load a malicious program file when starting the application.
- The OS could transfer control to a malicious signal handler when delivering a signal to the application.
- The OS could link in a malicious version of a dynamically loaded library (such as libc) used by the application.

Interrupted Program State Attacks

A malicious OS can attempt to modify or steal architectural state of an application while the application is not executing on the CPU. Examples include:

- Malicious system software could attempt to read interrupted program state to glean sensitive information from program register values saved to memory.
- Alternatively, it could modify interrupted program state (e.g., the PC) and put it back on to the processor on a return-from-interrupt or return-from-system call to redirect application execution to malicious code.

Attacks through System Services

A more subtle and complex class of attacks is possible through the higher-level (semantic) services an OS provides to applications [114, 36]. While these attacks are still not well understood, our solution addresses an important subset of them, namely, memory-based attacks via the `mmap` system call (the same subset also addressed by InkTag). Examples include the following:

- The OS is the source of randomness used by pseudorandom number generators to create random seeds, e.g., via the device /dev/random. The OS can compromise the degree of randomness and even give back the same random value on different requests, violating fundamental assumptions used for encrypting application data.

- The OS could return a pointer into the stack via `mmap` [36], thereby tricking the application into corrupting its stack to perform arbitrary code execution via return-to-libc [132] or return-oriented programming [118].
- The OS could grant the same lock to two different threads at the same time, introducing data races with unpredictable (and potentially harmful) results.

It is important to note that the *five categories of attack vectors* listed above are intended to be comprehensive but the specific examples within each category are not: there may be several other ways for the OS to attack an application within each category. Nevertheless, our solution enables applications to protect themselves against *all attacks* within the first four categories and a subset of attacks from the fifth.

7.3 Secure Computation Programming Model

The key feature we provide to a secure application is the ability to compute securely using secure memory, which we refer to as *ghost memory*, and to exchange data securely with external files and network interfaces. Applications do *not* have to be compiled with the SVA-OS compiler or instrumented in any particular way; those requirements only apply to the OS. In this section, we discuss the programmer's interface to secure computation. In Section 7.4, we show how our system, which we call *Virtual Ghost*, prevents the operating system from violating the integrity or confidentiality of an application that uses secure computation.

Name	**Description**
`allocgm(void * va, uintptr_t num)`	Map `num` page frames at the virtual address `va` (which must be within the ghost memory region).
`freegm(void * va, uintptr_t num)`	Free `num` page frames of ghost memory starting at `va` (which must be previously allocated via `allocgm`)

Table 7.1: Ghost Memory Management Instructions

7.3.1 Virtual Ghost Memory Organization

Virtual Ghost divides the address space of each process into three partitions. The first partition holds traditional, user-space application memory while a second partition, located at the high end of the virtual address space, holds traditional kernel-space memory. The kernel space memory is mapped persistently and is common across all processes, while the user-space memory mappings change as processes are switched

on and off the CPU. Operating systems such as Linux and BSD Unix already provide this kind of user-space/kernel-space partitioning [30, 99].

A new third partition, the ghost memory partition, is application-specific and is accessible only to the application and to Virtual Ghost. Physical page frames mapped to this partition hold application code, thread stacks, and any data used by secure computation. These page frames logically belong to the process and, like anonymous `mmap()` memory mappings, are unmapped from/mapped back into the virtual address space as the process is context switched on and off the processor.

Some applications may choose not to protect any of their memory, in which case the Ghost memory partition would go unused (their code and thread stacks would be mapped into ordinary application memory). Others may choose to protect *all* of their memory except a small portion used to pass data to or from the operating system. The latter configuration is essentially similar to what Overshadow provides, and Virtual Ghost provides this capability in the same way – by interposing wrappers on system calls. This is described briefly in Section 7.6.

7.3.2 Ghost Memory Allocation and Deallocation

An application wishing to execute securely without trusting the OS would obtain one or more chunks of ghost memory from Virtual Ghost; this memory is allocated and deallocated using two new "system calls" shown in Table 7.1. The instruction `allocgm()` asks Virtual Ghost to map one or more physical pages into the ghost partition starting at a specific virtual address. Virtual Ghost requests physical page frames from the operating system, verifies that the OS has removed all virtual to physical mappings for the frames, maps the frames starting at the specified address within the application's ghost partition, and zeroes out the frames' contents. The instruction `freegm()` tells Virtual Ghost that the block of memory at a specified virtual address is no longer needed and can be returned to the OS. Virtual Ghost unmaps the frames, zeroes their contents, and returns them to the operating system.

These instructions are not designed to be called directly by application-level code (although they could). A more convenient way to use these instructions is via a language's run-time library (e.g., the C standard library), which would use them to create language allocation functions that allocate ghost memory, e.g., using a modified version of `malloc`.

Note that ghost memory pages cannot be initialized using demand-paging from persistent storage into physical memory. Ghost Memory is like anonymous `mmap` memory, which Virtual Ghost can provide to an application at startup and when the application allocates it via `allocgm()`. To get data from the network or file system into ghost memory, the application must first read the data into traditional memory (which is OS

accessible) and then copy it (or decrypt it) into ghost memory. Other systems (e.g., InkTag [75]) perform the decrypt/copy operation transparently via the demand-paging mechanism. By requiring the application to decrypt data explicitly, Virtual Ghost avoids the complications of recovering encrypted application data after a crash (because the encryption keys are visible to the application and not hidden away within the Virtual Ghost VM). It also gives the application more flexibility in choosing different encryption algorithms and key lengths. Furthermore, this approach simplifies the design and reduces the size of Virtual Ghost, thus keeping Virtual Ghost's Trusted Computing Base (TCB) small.

7.3.3 I/O, Encryption, and Key Management

Applications that use ghost memory require secure mechanisms for communicating data with the external world, including local disk or across a network. Duplicating such I/O services in Virtual Ghost would significantly increase the size and complexity of the system's TCB. Therefore, like Overshadow [38] and InkTag [75], we let the untrusted OS perform all I/O operations. Applications running on Virtual Ghost must use cryptographic techniques (i.e., encryption, decryption, and digital signatures) to protect data confidentiality and detect potential corruption when writing data to or reading data from the OS during I/O.

Upon startup, applications need encryption keys to access persistently stored data (such as files on a hard drive) stored during previous executions of the application. Virtual Ghost provides each application with an application-specific public-private key pair that is kept secure from the OS and all other applications on the system. The application is responsible for encrypting (decrypting) secret data using this key pair before passing the data to (after receiving the data from) explicit I/O operations. Wrappers for widely used I/O operations such as `read()` and `write()` can make it largely transparent for applications to do the necessary encryption and decryption.

Using encryption for local IPC, network communication, and file system data storage allows applications to protect data confidentiality and to detect corruption. For example, to protect data confidentiality, an application can encrypt data with its public key before asking the OS to write the data to disk. To detect file corruption, an application can compute a file's checksum and encrypt and store the checksum in the file system along with the contents of the file. When reading the file back, it can recompute the checksum and validate it against the stored value. The OS, without the private key, cannot modify the file's contents and update the encrypted checksum with the appropriate value.

Unlike programmed I/O, swapping of ghost memory is the responsibility of Virtual Ghost. Virtual Ghost maintains its own public/private key pair for each system on which it is installed. If the OS indicates to Virtual Ghost that it wishes to swap out a ghost page, Virtual Ghost will encrypt and checksum the page with its keys before providing the OS with access. To swap a page in, the OS provides Virtual Ghost with the encrypted page contents; Virtual Ghost will verify that the page has not been modified and place it back into the ghost memory partition in the correct location. This design not only provides secure swapping but allows the OS to optimize swapping by first swapping out traditional memory pages.

7.3.4 Security Guarantees

An application that follows the guidelines above on a Virtual Ghost system obtains a number of strong guarantees, even in the presence of a hostile or compromised operating system or administrator account. All these guarantees apply to application data allocated in ghost memory. By “attacker" below, we mean an entity that controls either the OS or any process other than the application process itself (or for a multi-process or distributed application, the set of processes with which the application explicitly shares ghost memory or explicitly transfers the contents of that memory).

1. An attacker cannot read or write application data in memory or in CPU registers.

2. An attacker cannot read or write application code and cannot subvert application control flow at any point during the application execution, including application startup, signal delivery, system calls, or shutdown.

3. An attacker cannot read data that the application has stored unencrypted in ghost memory while the data is swapped out to external storage, nor can it modify such data without the corruption being detected before the data is swapped back into memory.

4. An attacker cannot read the application’s encryption key, nor can it modify the application’s encryption key or executable image undetected. Such modifications will be detected when setting the application up for execution and will prevent application startup.

5. By virtue of the application’s encryption key being protected, data encrypted by the application cannot be read while stored in external storage or in transit via I/O (e.g., across a network). Likewise, any corruption of signed data will be detected.

7.4 Enforcing Secure Computation

In this section, we show how Virtual Ghost is designed to prevent the operating system from violating the integrity or confidentiality of secure applications. Section 7.5 describes implementation details of this design.

7.4.1 Overview of Virtual Ghost

Our approach for protecting ghost memory is conceptually simple: we instrument (or "sandbox") memory access instructions in the kernel to ensure that they cannot access this memory. Virtual Ghost also instruments kernel code with control-flow integrity (CFI) checks to ensure that our sandboxing instrumentation is not bypassed [10, 160].

While sandboxing prevents attacks that attempt to access ghost memory directly, it does not prevent the other attacks described in Section 7.2.2. Our design additionally needs a way to restrict the interactions between the system software and both the hardware and applications. For example, our system must be able to verify (either statically or at run-time) that MMU operations will not alter the mapping of physical page frames used for ghost memory, and it must limit the types of changes that system software can make to interrupted program state.

To achieve these goals, our system needs a framework that provides both compiler analysis and instrumentation of operating system kernel code as well as a way of controlling interactions between the system software, the hardware, and applications. The Secure Virtual Architecture (SVA) framework (described in Chapter 3) meets these requirements, and we modify and extend it to support Virtual Ghost. A high-level overview of our approach, built on SVA, is shown in Figure 7.1.

7.4.2 Preventing Data Accesses in Memory

In this and the next few subsections, we discuss how Virtual Ghost addresses each of the five attack vectors described in Section 7.2.2.

Controlling Direct Memory Accesses

The SVM VM must perform two kinds of instrumentation when generating native code for the kernel (both the core kernel and dynamically loaded kernel modules). First, it must instrument loads and stores to prevent them from accessing ghost memory and the SVA VM internal memory. Recall that application pages are usually left mapped in the kernel's address space when entering the kernel on a system call, interrupt or trap.

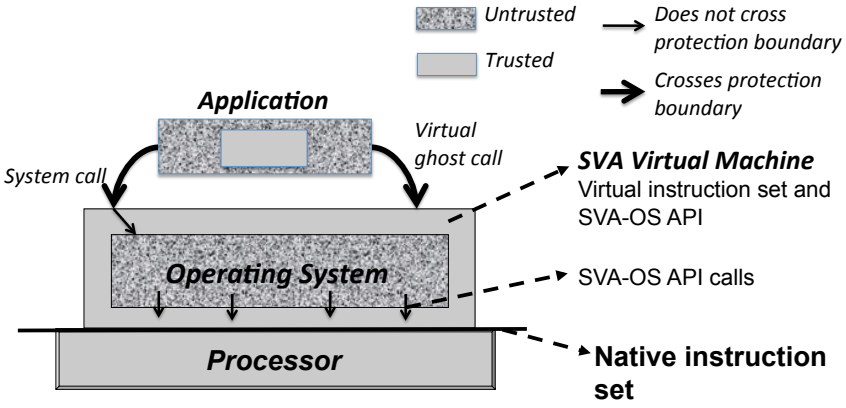


Figure 7.1: System Organization with Virtual Ghost

By instrumenting loads and stores, we avoid the need to unmap the ghost memory pages or (as Overshadow and Inktag do) to encrypt them before an OS access. Second, it must ensure that the instrumentation cannot be skipped via control-flow hijacking attacks [10].

By strategically aligning and sizing the SVA VM and ghost memory partitions, the load/store instrumentation can use simple bit-masking to ensure that a memory address is outside these partitions. The control-flow integrity instrumentation places checks on all returns and indirect calls to ensure that the computed address is a valid control-flow target. To support native code applications, our control-flow integrity checks also ensure that the address is within the kernel address space. A side benefit of our design is that the operating system kernel gets strong protection against control flow hijacking attacks.

MMU Protections

To prevent attacks that use illegal MMU page table entries, Virtual Ghost extends the MMU configuration instructions in SVA-OS to perform additional checks at run-time, which ensure that a new MMU configuration does not leave ghost memory accessible to the kernel. Specifically, Virtual Ghost does not permit the operating system to map physical page frames used by ghost memory into any virtual address. Virtual Ghost also prevents the operating system from modifying any virtual to physical page mapping for virtual addresses that belong to the ghost memory partition.

Virtual Ghost also protects other types of memory in addition to ghost memory. The SVA VM internal memory has the same restrictions as ghost memory. Native code is similarly protected; Virtual Ghost prevents native code pages from being remapped or made writable. This prevents the OS from bypassing the instrumentation or inserting arbitrary instructions into application code. Native code, unlike ghost memory or SVA internal memory, is made executable.

DMA Protections

Ghost Memory should never be the target of a legitimate DMA request. If Virtual Ghost can prevent DMA transfers to or from ghost physical page frames, then it can prevent DMA-based attacks.

SVA requires an IOMMU [12] and configures it to prevent I/O devices from writing into the SVA VM memory (discussed in Chapters 4 and 5). It further provides special I/O instructions for accessing processor I/O ports and memory-mapped I/O devices. Both SVA and Virtual Ghost must prevent the OS from reconfiguring the IOMMU to expose ghost memory to DMA transfers. How this is done depends on whether the hardware uses I/O ports or memory-mapped I/O to configure the IOMMU. If the hardware uses I/O ports, then SVA uses run-time checks within the I/O port read and write instructions as described in Chapter 5. If the hardware uses memory-mapped I/O, then SVA and Virtual Ghost simply use the MMU checks described above to prevent the memory-mapped physical pages of the IOMMU device from being mapped into the kernel or user-space virtual memory; instead, it is only mapped into the SVA VM memory, and the system software needs to use the I/O instructions to access it.

7.4.3 Preventing Data Accesses During I/O

Virtual Ghost relies on application-controlled encryption and hashing to prevent data theft or tampering during I/O operations, and uses automatic encryption and hashing done by the Virtual Ghost VM for page swapping, as described in Section 7.3.3.

The main design challenge is to find a way to start the application with encryption keys that cannot be compromised by a hostile OS. As noted earlier, Virtual Ghost maintains a public/private key pair for each system on which it is installed. We assume that a Trusted Platform Module (TPM) coprocessor is available; the storage key held in the TPM is used to encrypt and decrypt the private key used by Virtual Ghost. The application's object code file format is extended to contain a section for the application encryption keys, which are encrypted with the Virtual Ghost public key. Each time the application is started up, Virtual Ghost decrypts the encryption key section with the Virtual Ghost private key and places it into its internal

SVA VM memory before transferring control to the application. An application can use the `sva.getKey()` instruction to retrieve the key from the Virtual Ghost VM and store a copy in its ghost memory. If an application requires multiple private keys, e.g., for communicating with different clients, it can use its initial private key to encrypt and save these additional keys in persistent storage. Thus, Virtual Ghost and the underlying hardware together enable a *chain of trust* that cannot be compromised by the OS or other untrusted applications:

TPM storage key ⇒ Virtual Ghost private key ⇒ Application private key ⇒ Additional application keys.

The use of a separate section in the object code format allows easy modification of the keys by trusted tools. For example, a software distributor can place unique keys in each copy of the software before sending the software to a user. Similarly, a system administrator could update the keys in an application when the system is in single-user mode booted from trusted media.

7.4.4 Preventing Code Modification Attacks

Virtual Ghost prevents the operating system from loading incorrect code for an application, modifying native code after it is loaded, or repurposing existing native code instruction sequences for unintentional execution (e.g., via return-oriented programming).

To prevent the system software from loading the wrong code for an application, Virtual Ghost assumes that the application is installed by a trusted system administrator (who may be local or remote). The application's executable, including the code section and embedded application key described in Section 7.3.3, is signed by Virtual Ghost's public key when the application binary is installed. If the system software attempts to load different application code with the application's key, Virtual Ghost refuses to prepare the native code for execution.

To prevent native code modification in memory, Virtual Ghost ensures that the MMU maps all native code into non-writable virtual addresses. It also ensures that the OS does not map new physical pages into virtual page frames that are in use for OS, SVA-OS, or application code segments.

Virtual Ghost prevents repurposing existing instruction sequences or functions simply because the Control Flow Integrity (CFI) enforcement (for ensuring that sandboxing instructions are not bypassed) prevents all transfers of control not predicted by the compiler. For example, a buffer overflow in the kernel could overwrite a function pointer, but if an indirect function call using that function pointer attempted to go to any location other than one of the predicted callees of the function, the CFI instrumentation would detect that and terminate the execution of the kernel thread.

Finally, Virtual Ghost also prevents the OS from subverting signal dispatch to the application, e.g., by executing arbitrary code instead of the signal handler in the application context. Although this is essentially a code modification attack, the mechanisms used to defend against this are based on protecting interrupted program state, described next.

7.4.5 Protecting Interrupted Program State

An attacker may attempt to read interrupted program state (the program state that is saved on a system call, interrupt, or trap) to glean confidential information. Alternatively, she may modify the interrupted program state to mount a control-hijack or non-control data attack on the application. Such an attack could trick the application into copying data from ghost memory into traditional memory where the operating system can read or write it. Note that such an attack works even on a completely memory-safe program. We have implemented such an attack as described in Section 7.7.

The SVA framework (and hence Virtual Ghost) calls this interrupted program state the *Interrupt Context*. The creation and maintenance of the Interrupt Context is performed by the SVA virtual machine. While most systems save the Interrupt Context on the kernel stack, Virtual Ghost saves the Interrupt Context within the SVA VM internal memory. Virtual Ghost also zeros out registers (except registers passing system call arguments for system calls) after saving the Interrupt Context but before handing control over to the OS. With these two features, the OS is unable to read or modify the Interrupt Context directly or glean its contents from examining current processor state.

The OS does need to make controlled changes to the Interrupt Context. For example, process creation needs to initialize a new Interrupt Context to return from the `fork()` system call [30], and signal handler dispatch needs to modify the application program counter and stack so that a return-from-interrupt instruction will cause the application to start executing its signal handler [30, 99].

SVA provides instructions for manipulating the Interrupt Context, and Virtual Ghost enhances the checks on them to ensure that they do not modify the Interrupt Context in an unsafe manner, as explained below.

Secure Signal Handler Dispatch

Virtual Ghost provides instructions for implementing secure signal handler dispatch. Signal handler dispatch requires saving the Interrupt Context, modifying the interrupted program state so that the signal handler is invoked when the interrupted application is resumed, and then reloading the saved Interrupt Context back into the interrupted program state buffer when the `sigreturn()` system call is called.

Virtual Ghost pushes and pops a copy of the Interrupt Context on and off a per-thread stack within the SVA VM internal memory, whereas the original SVA-M described in Chapters 4 and 5 allowed the kernel to make a copy of the Interrupt Context in the kernel heap for unprivileged applications (because SVA-M did not aim to protect application control flow from the OS, whereas Virtual Ghost does). This enhancement to the original SVA design ensures that the OS cannot modify the saved state *and* ensures that the OS restores the correct state within the correct thread context.

SVA provides an operation, `sva.ipush.function()`, which the operating system can use to modify an Interrupt Context so that the interrupted program starts execution in a signal handler. The OS passes in a pointer to the application function to call and the arguments to pass to this function, and Virtual Ghost modifies the Interrupt Context on the operating system's behalf. For efficiency, we allow `sva.ipush.function()` to modify the application stack even though the stack may be within ghost memory. Since Virtual Ghost only adds a function frame to the stack, it cannot read or overwrite data that the application is using.

To ensure that the specified function is permissible, Virtual Ghost provides an operation called `sva.permitFunction()` which the *application* must use to register a list of functions that can be "pushed"; `sva.ipush.function()` refuses to push a function that is not in this list. To simplify application development, we provide wrappers for the `signal` and `sigaction` system calls, which register the signal handlers transparently, without needing to modify the application.

Secure Process Creation

A thread is composed of two pieces of state: an Interrupt Context, which is the state of an interrupted user-space program, and a kernel-level processor state that Virtual Ghost calls *Thread State* that represents the state of the thread before it was taken off the CPU.[2] Creating a new thread requires creating a new Interrupt Context and Thread State.

While commodity operating systems create these structures manually, Virtual Ghost provides a single function, `sva.init.icontext()`, to create these two pieces of state. This function creates these new state structures within the SVA VM internal memory to prevent tampering by the system software. The newly created Interrupt Context is a clone of the Interrupt Context of the current thread. The new Thread State is initialized so that, on the next context switch, it begins executing in a function specified by the operating system. In order to maintain kernel control-flow integrity, Virtual Ghost verifies that the specified function is the entry point of a kernel function.

[2]SVA-M divided Thread State into Integer State and Floating Point State as an optimization. Virtual Ghost does not.

Any ghost memory belonging to the current thread will also belong to the new thread; this transparently makes it appear that ghost memory is mapped as shared memory among all threads and processes within an application.

Executing a new program (e.g., via the `execve()` system call) also requires reinitializing the Interrupt Context [30]: the program counter and stack pointer must be changed to execute the newly loaded program image, and, in the case of the first user-space process on the system (e.g., `init`), the processor privilege level must be changed from privileged to unprivileged. Virtual Ghost also ensures that the program counter points to the entry of a program that has previously been copied into SVA VM memory; Section 7.4.4 describes how Virtual Ghost ensures that this program has not tampered by the OS before or during this copy operation.

Finally, any ghost memory associated with the interrupted program is unmapped when the Interrupt Context is reinitialized. This ensures that newly loaded program code does not have access to the ghost memory belonging to the previously executing program.

7.4.6 Mitigating System Service Attacks

System service attacks like those described in Section 7.2.2 are not yet well understood [36]. However, Virtual Ghost provides some protection against such attacks that are known.

First, Virtual Ghost employs an enhanced C/C++ compiler that instruments system calls in ghosting applications to ensure that pointers passed into or returned by the operating system are not pointing into ghost memory. This instrumentation prevents accidental overwrites of ghost memory. This protects private data and prevents stack-smashing attacks (because the stack will be located in ghost memory).

Second, the Virtual Ghost VM provides an instruction for generating random numbers. The random number generator is built into the Virtual Ghost VM and can be trusted by applications for generating random numbers. This defeats Iago attacks that feed non-random numbers to applications [36].

While these protections are far from comprehensive, they offer protection against existing system service attacks.

7.5 Implementation

We created Virtual Ghost by modifying the 64-bit implementation of SVA described in Chapter 3 and used the FreeBSD 9.0 kernel that was ported to SVA. The Virtual Ghost instructions are implemented in the SVA-OS run-time library that is linked into the kernel.

Virtual Ghost builds on, modifies, and adds a number of SVA-OS operations to implement the design described. We briefly summarize them here, omitting SVA-OS features that are needed for hosting a kernel and protecting SVA itself, but not otherwise used by Virtual Ghost for application security. We first briefly describe the compiler instrumentation and end with a brief summary of features that are not yet implemented.

Compiler Instrumentation: The load/store and CFI instrumentation is implemented as two new passes in the LLVM 3.1 compiler infrastructure. The load/store instrumentation pass transforms code at the LLVM IR level; it instruments mid-level IR loads, stores, atomic operations, and calls to `memcpy()`. The CFI instrumentation pass is an updated version of the pass written by Zeng et. al. [160] that works on x86_64 machine code. It analyzes the machine code that LLVM 3.1 generates and adds in the necessary CFI labels and checks. It also masks the target address to ensure that it is not a user-space address. We modified the Clang/LLVM 3.1 compiler to use these instrumentation passes when compiling kernel code. To avoid link-time interprocedural analysis, which would be needed for precise call graph construction, our CFI instrumentation uses a very conservative call graph: we use one label both for call sites (i.e., the targets of returns) and for the first address of every function. While conservative, this call graph allows us to measure the performance overheads and should suffice for stopping advanced control-data attacks.

We placed the ghost memory partition into an unused 512 GB portion of the address space (`0xffffff0000000000` – `0xffffff8000000000`). The load/store instrumentation determines whether the address is greater than or equal to `0xffffff0000000000` and, if so, ORs it with 2^{39} to ensure that the address will not access ghost memory. While our design would normally use some of this 512 GB partition for SVA internal memory, we opted to leave the SVA internal memory within the kernel's data segment; we added new instrumentation to kernel code, which changes an address to zero before a load, store, or `memcpy()` if it is within the SVA internal memory. This additional instrumentation adds some overhead but simplifies development.

To defend against Iago attacks through the `mmap` system call, a separate mid-level LLVM IR instrumentation pass performs identical bit-masking instrumentation to the return values of `mmap()` system calls for user-space application code.[3] This instrumentation moves any pointer returned by the kernel that points into ghost memory out of ghost memory. In this way, Iago attacks using `mmap()` [36] cannot trick an application into writing data into its own ghost memory. If an application stores its stack and function pointers in ghost memory, then our defense should prevent Iago attacks from subverting application control flow integrity.

[3] An alternative implementation method would be to implement a C library wrapper function for `mmap()`.

Memory Management: SVA-OS provides operations that the kernel uses to insert and remove page table entries at each level of a multi-level page table and ensures that they do not violate internal consistency of the page tables or the integrity of SVA code and data. Virtual Ghost augments these operations to also enforce the MMU mapping constraints described in Section 7.4.2.

Launching Execution: Virtual Ghost provides the operation, `sva.reinit.icontext()`, to *reinitialize* an Interrupt Context with new application state. This reinitialization will modify the program counter and stack pointer in the Interrupt Context so that, when resumed, the program will begin executing new code.

Virtual Ghost uses the x86_64 Interrupt Stack Table (IST) feature [4] to protect the Interrupt Context. This feature instructs the processor to always change the stack pointer to a known location on traps or interrupts regardless of whether a change in privilege mode occurs. This allows the SVA VM to direct the processor to save interrupted program state (such as the program counter) within the SVA VM internal memory so that it is never accessible to the operating system.

Signal Delivery: Virtual Ghost implements the `sva.icontext.save()` and `sva.icontext.load()` instructions to save and restore the Interrupt Context before and after signal handler dispatch. These instructions save the Interrupt Context within SVA memory to ensure that the OS cannot read or write it directly.

What Is Not Yet Implemented: Our implementation so far does not include a few features described previously. (1) While explicit I/O is supported, the key management functions are only partially implemented: Virtual Ghost does not provide a public/private key pair; does not use a TPM; and application-specific keys are not embedded in application binaries (instead, a 128-bit AES application key is hard-coded into SVA-OS for our experiments). (2) Swapping of ghost memory is not implemented. (3) The system does not yet include the DMA protections. We believe that IOMMU configuration is rare, and therefore, the extra protections for DMA should not add noticeable overhead. (4) Finally, some operations in the FreeBSD kernel are still handled by inline assembly code (e.g., memory-mapped I/O loads and stores), but we do not believe that these unported operations will significantly affect performance once they are ported properly.

Trusted Computing Base: Since the Virtual Ghost implementation is missing a few features, measuring the size of its Trusted Computing Base (TCB) is premature. However, the vast majority of the functionality has been implemented, and the current code size is indicative of the approximate size of the TCB. Virtual Ghost currently includes only 5,344 source lines of code (ignoring comments, whitespace, etc.). This count

includes the SVA VM run-time system and the passes that we added to the compiler to enforce our security guarantees. Overall, we believe that the complexity and attack surface of Virtual Ghost are far smaller than modern production hypervisors like XenServer but approximately comparable to a minimal hypervisor.

7.6 Securing OpenSSH

To demonstrate that our system can secure real applications, we modified three programs from the OpenSSH 6.2p1 application suite to use ghost memory: `ssh`, `ssh-keygen`, and `ssh-agent`. The `ssh-keygen` program generates public/private key pairs which `ssh` can use for password-less authentication to remote systems. The `ssh-agent` server stores private encryption keys which the `ssh` client may use for public/private key authentication. We used a single private "application key" for all three programs so that they could share encrypted files.

We modified `ssh-keygen` to encrypt all the private authentication key files it generates with the application key; our `ssh` client decrypts the authentication keys with the application private key upon startup and places them, along with all other heap objects, into ghost memory. Since the OS cannot gain access to the application key, it cannot decrypt the authentication keys that are stored on disk, and it cannot read the cleartext versions out of `ssh`'s or `ssh-keygen`'s ghost memory.

We modified the FreeBSD C library so that the heap allocator functions (`malloc()`, `calloc()`, `realloc()`) allocate heap objects in ghost memory instead of in traditional memory; the changes generate a 216 line patch. To ease porting, we wrote a 667-line system call wrapper library that copies data between ghost memory and traditional memory as necessary. This wrapper library also provides wrappers for `signal()` and `sigaction()` that register the signal handler functions with Virtual Ghost before calling the kernel's `signal()` and `sigaction()` system calls. The compiler and linker did not always resolve system calls to our wrapper functions properly, so we made some manual modifications to the programs. We also modified the programs to use traditional memory (allocated via `mmap()`) to store the results of data to be sent to `stdout` and `stderr` to reduce copying overhead. In total, our changes to OpenSSH can be applied with a patch that adds 812 and deletes 68 lines of code (OpenSSH contains 9,230 source lines of code).

We tested our applications on the system used for our experiments (see Section 7.8). We used `ssh-keygen` to generate a new private and public key for DSA authentication; the generated private key was encrypted while the public key was not encrypted. We then installed the public key on another system and used `ssh` to log into that system using DSA authentication.

For `ssh-agent`, we added code to place a secret string within a heap-allocated memory buffer. The rootkit attacks described in Section 7.7 attempt to read this secret string. The goal of adding this secret string is that `ssh-agent` treats it identically to an encryption key (it can use the string internally but never outputs it to another program). We used a secret string as it is easier to find in memory and easier to identify as the data for which our attack searches.

Our enhanced OpenSSH application suite demonstrates that Virtual Ghost can provide security-critical applications with in-memory secrets (e.g., the keys held by `ssh-agent`) and secure, long-term storage (e.g., the authentication keys created by `ssh-keygen` and read by `ssh`). It also demonstrates that a suite of cooperating applications can securely share data on a hostile operating system via a shared application key.

7.7 Security Experiments

To evaluate the security of our system, we built a malicious kernel module that attempts to steal sensitive information from a victim process. This module, based on the code from Joseph Kong's book [84], can be configured by a non-privileged user to mount one of two possible attacks – direct memory access or code injection – on a given victim process. The malicious module replaces the function that handles the `read()` system call and executes the attack as the victim process reads data from a file descriptor.

In the first attack, the malicious module attempts to directly read the data from the victim memory and print it to the system log.

In the second attack, the malicious module attempts to make the victim process write the confidential data out to a file. The attack first opens the file to which the data should be written, allocates memory in the process's address space via `mmap()`, and copies exploit code into the memory buffer. The attack then sets up a signal handler for the victim process that calls the exploit code. The malicious module then sends a signal to the victim process, triggering the exploit code to run in the signal handler. The exploit code copies the data into the `mmap`'ed memory and executes a `write()` system call to write the secret data out to the file opened by the malicious module.

We used both attacks on our ssh-agent program, described in Section 7.6. When we install the malicious module without instrumenting its code and run ssh-agent with `malloc()` configured to allocate memory objects in traditional memory, both attacks succeed.

We then recompiled our malicious module using our modified Clang compiler to insert the instrumentation required for Virtual Ghost. We reran both attacks on ssh-agent with `malloc()` configured to allocate memory objects in ghost memory. The first attack fails because the load/store instrumentation changes the pointer

in the malicious module to point outside of ghost memory; the kernel simply reads unknown data out of its own address space. The second attack is thwarted because `sva.ipush.function()` recognizes that the exploit code isn't one of the functions registered as a valid target of `sva.ipush.function()`.

Note that a number of other possible attacks from within kernel code *are simply not expressible in our system*, e.g., anything that requires using assembly code in the kernel (such as to access CPU registers), or manipulating the application stack frames, or modifying interrupted application state when saved in memory. The second attack above illustrates that much more sophisticated multi-step exploits are needed to get past the basic protections against using assembly code or accessing values saved in SVA VM memory or directly reading or writing application code and data (as in the first attack). Virtual Ghost is successfully able to thwart even this sophisticated attack.

7.8 Performance Experiments

We ran our performance experiments on a Dell Precision T1650 workstation with an Intel® Core™ i7-3770 processor at 3.4 GHz with 8 MB of cache, 16 GB of RAM, an integrated PCIE Gigabit Ethernet card, a 7200 RPM 6 Gb/s 500 GB SATA hard drive, and a 256 GB Solid State Drive (SSD). Files in `/usr` were stored on the SSD. For network experiments, we used a dedicated Gigabit Ethernet network. The client machine was an iMac with a 4-core hyper-threaded Intel® Core™ i7 processor at 2.6 GHz with 8 GB of RAM.

We evaluated our system's performance on microbenchmarks as well as a few applications. We used microbenchmarks to see how Virtual Ghost affects primitive OS operations, `thttpd` and `sshd` for network applications, and Postmark [116] for a file system intensive program.

We conducted our experiments by booting the machine into single-user mode to minimize noise from other processes running on the system. Our baseline, unless otherwise noted, is a native FreeBSD kernel compiled with the LLVM 3.1 compiler and configured identically to our Virtual Ghost FreeBSD kernel.

7.8.1 Microbenchmarks

In order to understand how our system affects basic OS performance, we measured the latency of various system calls using LMBench [100] from the FreeBSD 9.0 ports tree. For those benchmarks that can be configured to run the test for a specified number of iterations, we used 1,000 iterations. Additionally, we ran each benchmark 10 times.

As Table 7.2 shows, our system can add considerable overhead to individual operations. System call entry increases by 3.9 times. We compare these relative slowdowns with InkTag, which has reported LMBench

Test	Native	Virtual Ghost	Overhead	InkTag
null syscall	0.091	0.355	3.90x	55.8x
open/close	2.01	9.70	4.83x	7.95x
mmap	7.06	33.2	4.70x	9.94x
page fault	31.8	36.7	1.15x	7.50x
signal handler install	0.168	0.545	3.24x	-
signal handler delivery	1.27	2.05	1.61x	-
fork + exit	63.7	283	4.40x	5.74x
fork + exec	101	422	4.20x	3.04x
select	3.05	10.3	3.40x	-

Table 7.2: LMBench Results. Time in Microseconds.

File Size	Native	Virtual Ghost	Overhead
0 KB	166,846	36,164	4.61x
1 KB	116,668	25,817	4.52x
4 KB	116,657	25,806	4.52x
10 KB	110,842	25,042	4.43x

Table 7.3: LMBench: Files Deleted Per Second.

results as well [75]. Our slowdowns are nearly identical to or better than InkTag on 5/7 microbenchmarks: all except `exec()` and file deletion/creation. Our file deletion/creation overheads (shown in Tables 7.3 and 7.4) average 4.52x and 4.94x, respectively, across all file sizes, which is slower than InkTag. System calls and page faults, two of the most performance critical OS operations, are both considerably faster on Virtual Ghost than on InkTag.

While Virtual Ghost adds overhead, it provides double benefits for the cost: in addition to ghost memory, it provides protection to the OS itself via control flow integrity.

7.8.2 Web Server Performance

We used a statically linked, non-ghosting version of the `thttpd` web server [115] to measure the impact our system had on a web server. We used ApacheBench [1] to measure the bandwidth of transferring files between 1 KB and 1 MB in size. Each file was generated by collecting random data from the `/dev/random`

File Size	Native	Virtual Ghost	Overhead
0 KB	156,276	33,777	4.63x
1 KB	97,839	18,796	5.21x
4 KB	97,102	18,725	5.19x
10 KB	85,319	18,095	4.71x

Table 7.4: LMBench: Files Created Per Second.

device and stored on the SSD. We configured ApacheBench to make 100 simultaneous connections and to perform 10,000 requests for the file for each run of the experiment. We ran each experiment 20 times.

Figure 7.2 shows the mean performance of transferring a file of each size and displays the standard deviation as error bars. The data show that the impact of Virtual Ghost on the Web transfer bandwidth is negligible.

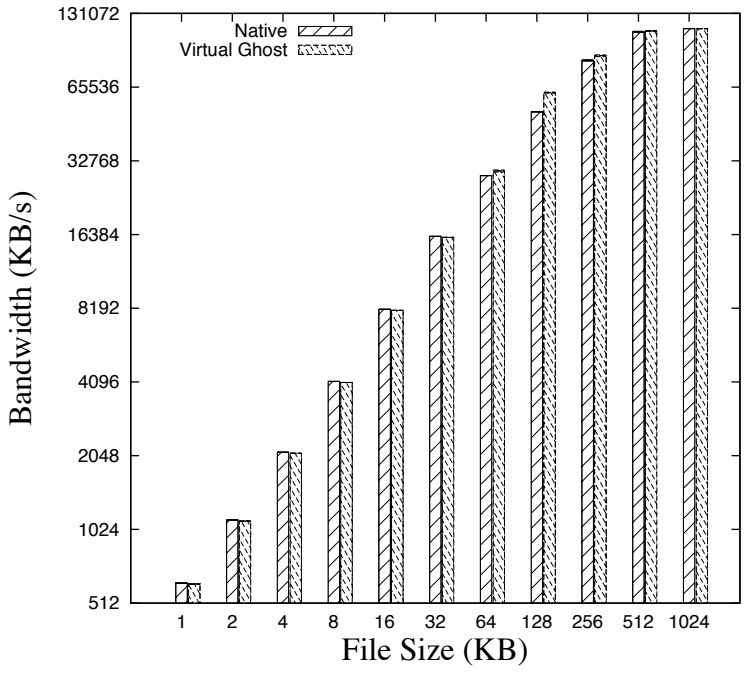


Figure 7.2: Average Bandwidth of thttpd

7.8.3 OpenSSH Performance

To study secure bulk data transfer performance, we measured the bandwidth achieved when transferring files of various sizes using the OpenSSH server and client [117]. We measured the performance of the `sshd` server without ghosting and our ghosting `ssh` client described in Section 7.6. Files were created using the same means described in Section 7.8.2.

Server Performance Without Ghosting

We ran the pre-installed OpenSSH server on our test machine and used the standard Mac OS X OpenSSH `scp` client to measure the bandwidth achieved when transferring files. We repeated each experiment 20 times and report standard deviation bars. The baseline system is the original FreeBSD 9.0 kernel compiled with Clang and configured identically to our Virtual Ghost FreeBSD kernel.

Figure 7.3 shows the mean bandwidth for the baseline system and Virtual Ghost. We observe bandwidth reductions of 23% on average, with a worst case of 45%, and negligible slowdowns for large file sizes.

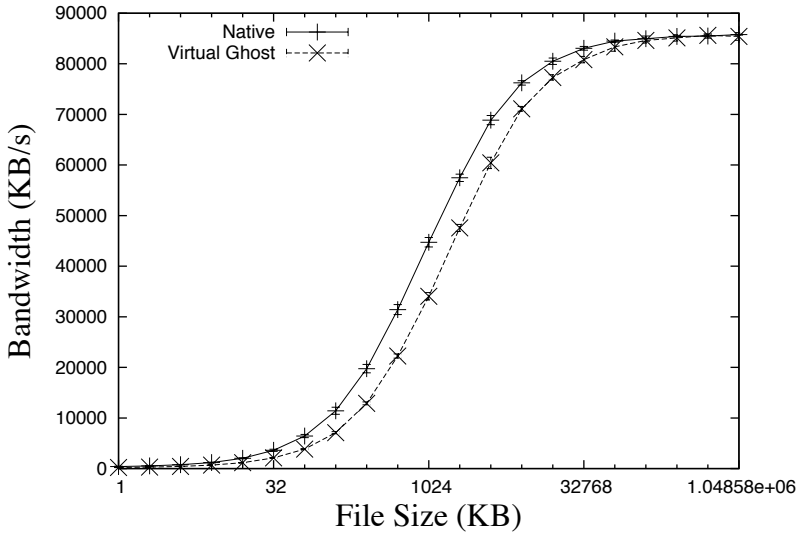


Figure 7.3: SSH Server Average Transfer Rate

7.8.4 Client Performance With Ghosting

To measure the effect of using ghost memory, we measured the average bandwidth of transferring file sizes of 1 KB to 1 MB using both the unmodified OpenSSH `ssh` client and our ghosting `ssh` client described in Section 7.6. We transferred files by having `ssh` run the `cat` command on the file on the server. We ran both on the Virtual Ghost FreeBSD kernel to isolate the performance differences in using ghost memory. We

transferred each file 20 times. Figure 7.4 reports the average of the 20 runs for each file size as well as the standard deviation using error bars. (The numbers differ from those in Figure 7.3 because this experiment ran the ssh client on Virtual Ghost, instead of the sshd server.)

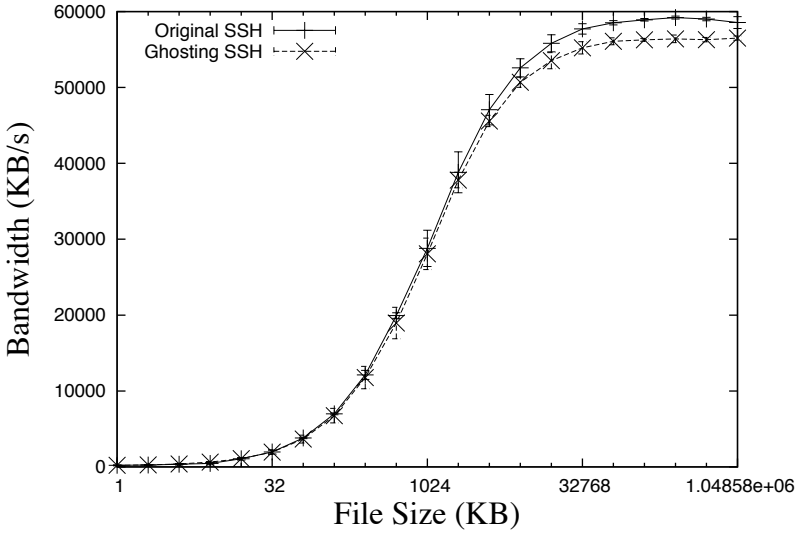


Figure 7.4: Ghosting SSH Client Average Transfer Rate

As Figure 7.4 shows, the performance difference between using traditional memory, which is accessible to the OS, and ghost memory is small. The maximum reduction in bandwidth by the ghosting `ssh` client is 5%.

7.8.5 Postmark Performance

In order to test a file system intensive benchmark, we ran Postmark [116]. Postmark mimics the behavior of a mail server and exercises the file system significantly.

Native (s)	Std. Dev.	Virtual Ghost (s)	Std. Dev.	Overhead
14.30	0.46	67.50	0.50	4.72x

Table 7.5: Postmark Results

We configured Postmark to use 500 base files with sizes ranging from 500 bytes to 9.77 KB with 512 byte block sizes. The read/append and create/delete biases were set to 5, and we configured Postmark to use buffered file I/O. All files were stored on the SSD. Each run of the experiment performed 500,000 transactions.

We ran the experiment 20 times on both the native FreeBSD kernel and the Virtual Ghost system. Table 7.5 shows the results. As Postmark is dominated by file operations, the slowdown of 4.7x is similar to the LMBench open/close system call overhead of 4.8x. Due to its file system intensive behavior, Postmark represents the worst case for the application benchmarks we tried.

7.9 Related Work

Several previous systems attempt to protect an application's code and data from a malicious operating system. Systems such as Overshadow [38, 114], SP3 [157], and InkTag [75] build on a full-scale commercial hypervisor (e.g., VMWare Server or XenServer). The hypervisor presents an encrypted view of the application's memory to the OS and uses digital signing to detect corruption of the physical pages caused by the OS. These systems do *not* prevent the OS from reading or modifying the encrypted pages. To simplify porting legacy applications, such systems include a shim library between the application and the operating system that encrypts and decrypts data for system call communication.

Hypervisor-based approaches offer a high level of compatibility with existing applications and operating systems, but suffer high performance overhead. Additionally, they add overhead to the common case (when the kernel reads/writes application memory correctly via the system call API). They also do not provide additional security to the operating system and do not compose as cleanly with other kernel security solutions that use hypervisor-based approaches.

Virtual Ghost presents a different point in the design space for protecting applications from an untrusted OS. First, it uses compiler instrumentation ("sandboxing" and control-flow integrity) instead of page protections to protect both application ghost memory pages as well as its own code and metadata from the OS (similar to systems such as SPIN [28], JavaOS [122], and Singularity [77, 64]). Second, it completely prevents the OS from reading and writing ghost memory pages rather than allowing access to the pages in encrypted form. Third, although Virtual Ghost introduces a hardware abstraction layer (SVA-OS) that is somewhat similar to a (minimal) hypervisor, SVA-OS does not have higher privilege than the OS; instead, it appears as a library of functions that the OS kernel code can call directly. Fourth, system calls from the secure application to the OS need not incur encryption and hashing overhead for non-secure data. Fifth, Virtual Ghost gives the application considerably more control over the choices of encryption and hashing keys, and over what subset of data is protected. Finally, the compiler approach hardens the OS against external exploits because it prevents both injected code and (by blocking control-flow hijacking) exploits that use existing code, such as return-oriented programming or jump-oriented programming. Moreover, the

compiler approach can be directly extended to provide other compiler-enforced security policies such as the comprehensive memory safety policy described in Chapters 4 and 5.

In addition to isolation, InkTag [75] also provides some valuable usability improvements for secure applications, including services for configuring access control to files. These features could be provided by SVA-OS in very similar ways, at the cost of a non-trivial relative increase in the TCB size.

Other recent efforts, namely TrustVisor [96], Flicker [97], and Memoir [113], provide special purpose hypervisors that enable secure execution and data secrecy for pieces of application logic. They obtain isolated execution of code and data secrecy via hardware virtualization. These approaches prevent the protected code from interacting with the system, which limits the size of code regions protected, and data must be sealed via trusted computing modules between successive invocations of the secure code. In contrast, Virtual Ghost provides continuous isolation of code and data from the OS without the need for secure storage or monopolizing system wide execution of code.

Several systems provide hardware support to isolate applications from the environment, including the OS. The XOM processor used by XOMOS [91] encrypts all instructions and data transferred to or from memory and can detect tampering of the code and data, which enables a secure application on XOMOS to trust nothing but the processor (not even the OS or the memory system). HyperSentry [26] uses server-class hardware for system management (IPMI and SMM) to perform stealthy measurements of a hypervisor without using software at a higher privilege level, while protecting the measurements from the hypervisor. These mechanisms are designed for infrequent operations, not for more extensive computation and I/O.

ARM Security Extensions (aka TrustZone) [24] create two virtual cores, operating as two isolated "worlds" called the Secure World and the Normal World, on a single physical core. Applications running in the Secure World are completely isolated from an OS and applications running in the Normal World. Secure World applications can use peripheral devices such as a keyboard or display securely. Virtual Ghost and any other pure-software scheme would need complex additional software to protect keyboard and display I/O. Intel Software Guard Extensions (SGX) provide isolated execution zones called enclaves that are protected from privileged software access including VMMs, OS, and BIOS [98]. The enclave is protected by ISA extensions and hardware access control mechanisms and is similar to Virtual Ghost in that it protects memory regions from the OS. Additionally, SGX provides both trusted computing measurement, sealing, and attestation mechanisms [19]. Unlike TrustZones and SGX, Virtual Ghost requires no architectural modifications and could provide a secure execution environment on systems that lack such hardware support.

7.10 Summary

Virtual Ghost provides application security in the face of untrusted operating systems. Virtual Ghost does not require a higher-privilege layer, as hypervisors do; instead, it applies compiler instrumentation combined with runtime checks on operating system code to provide ghost memory to applications. We ported a suite of real-world applications to use ghost memory and found that Virtual Ghost protected the applications from advanced kernel-level malware. We observe comparable performance, and much better in some instances, than the state of the art in secure execution environments, while also providing an additional layer of protection against external attacks on the operating system kernel itself.

Chapter 8

Conclusions and Future Work

This work presented the Secure Virtual Architecture (SVA): a compiler-based virtual machine that is capable of enforcing strong security policies on both application and kernel code. This work described how SVA abstracts the hardware so that compiler techniques can identify and control software/hardware interactions and application state manipulation, and it demonstrates how this control can be used to protect the operating system kernel (via memory safety and control-flow integrity enforcement) and applications (by restricting the operating system's ability to manipulate application data and control-flow).

While the current work is a promising start in providing strong security for real-world commodity systems, it is only the beginning: important research work in computer security that SVA can aid remains.

First, the security guarantees that SVA can provide to operating system kernel code and the speed with which it provides it can be improved. For example, the type-safety and sound points-to analysis described in Chapter 4 uses unification-based points-to analysis [88]. Future work can explore using a more precise points-to analysis algorithm (such as Anderson's [20]) to provide both tighter constraints on the subset of memory objects which a load or store may access as well as finding greater amounts of type-safety to exploit for improving performance.

Similarly, there are some memory safety issues that this work has not addressed. For example, certain optimizations may not be safe in a multi-threaded environment (e.g., hoisting a pointer arithmetic check out of a loop) because memory objects may be freed and the memory re-allocated by other processors. Likewise, there is a time-of-check-to-time-of-use issue in which a memory object may be freed between the time a pointer is checked and the time that it is used in a load or store. These issues stem from the asynchronous nature of the kernel and the fact that memory allocation and deallocation can change memory object bounds (fortunately, memory allocation and deallocation is the only way that an object's size can change due to asynchronous behavior).

Second, with its virtual instruction set, SVA lends itself well to formal analysis of its design and implementation. For example, we can expand the SVA small-step semantics described in Chapter 6 to support

all of the features of SVA. A complete set of semantics could provide a formal description of the system to those wishing to implement it on other processors or to those wishing to study its properties formally. Such a formal study might prove that SVA provides sound points-to analysis and callgraph analysis to the operating system kernel as claimed in Chapter 5. This proof would show that memory safety optimizations requiring points-to analysis are safe when performed by the SVA virtual machine. It would also ensure that formal verification techniques utilizing callgraph and points-to analysis are sound when used on commodity operating system kernels ported to SVA [57].

Third, the work on protecting applications from compromised operating systems is still in its infancy. Ghost memory, discussed in Chapter 7, protects application data from attack while it is in memory. However, applications still need to employ cryptographic techniques to protect their data when sending it through the operating system. While encryption and digital signatures can be used to protect confidentiality and to detect direct corruption, it is not yet clear how to detect more subtle attacks such as replay attacks (e.g., substituting a file's contents with older contents signed with the same key) or whether Virtual Ghost provides sufficient features to applications to detect such subtle attacks.

Finally, with its ability to analyze and instrument kernel code, we believe that SVA can enforce security policies in addition to those described in this dissertation. For example,

- SVA could be used to enforce global, system-wide information flow policies since it can transform application and OS kernel code and can analyze the interactions between the two.
- SVA could be used to detect integer overflow errors and to take action when they occur.
- SVA could be used to enforce locking disciplines to prevent deadlock and data races.

In summary, we believe that SVA can provide a foundation on which future security solutions for both application and operating system code can be built.

Appendix A

Secure Virtual Architecture Instruction Set

A.1 I/O

A.1.1 sva_io_read

Synopsis

integer sva_io_read(*ioptr* address)

Description

Read an input from a virtual I/O address.

Arguments

- address - The virtual I/O address from which to read.

Return Value

The value read from the I/O address is returned.

Virtual Ghost Checks

- The value `address` cannot lie within the ghost address space.

A.1.2 sva_io_write()

Synopsis

void sva_io_write(*integer* value, *ioptr* address)

Description

Output a value to a virtual I/O address.

Arguments

- value - The value to output to the device.
- address - The virtual I/O address to which to write.

Return Value

None.

CFI Checks

- The value `address` cannot lie within the ghost address space.

Virtual Ghost Checks

- The value `address` cannot lie within the ghost address space.

A.2 Interrupts

A.2.1 sva_load_lif

Synopsis

void sva_load_lif(*bool* enable)

Description

Enable or disable processor interrupts.

Arguments

- enable - If true, enables interrupts. Otherwise, disables interrupts.

Return Value

None

A.2.2 sva_save_lif

Synopsis

bool sva_save_lif(*void*)

Description

Return the value of interrupt enable flag.

Arguments

None.

Return Value

- true - Interrupts were enabled.
- false - Interrupts were disabled.

A.3 Event Handlers

A.3.1 sva_register_syscall

Synopsis

void sva_register_syscall(*unsigned* index, *unsigned* (*handler)(*unsigned* index, ...))

Description

Register a function to handle a system call.

Arguments

- index - The system call number for which the handler should be called. This value is passed to the handler when the handler is called.
- handler - A pointer to the kernel function that should be called when the system call specified by *index* is executed.

Return Value

None.

CFI Checks

- The function `handler` should be a valid kernel function pointer.

Notes

Interrupts will be disabled when control is passed to the specified handler function. It is the handler function's responsibility to re-enable interrupts if desired.

A.3.2 sva_register_interrupt

Synopsis

void sva_register_interrupt(*unsigned* index, *void* (*handler)(*unsigned* index))

Description

Register a function to handle an interrupt.

Arguments

- index - The interrupt vector number for which the handler should be called. This value is passed to the handler when the handler is called.
- handler - A pointer to the kernel function that should be called when the specified interrupt occurs.

Return Value

None.

CFI Checks

- The function `handler` should be a valid kernel function pointer.

Notes

Interrupts will be disabled when control is passed to the specified handler function. It is the handler function's reponsibility to re-enable interrupts if desired.

A.3.3 sva_register_general_exception

Synopsis

void sva_register_general_exception(*unsigned* index, *void* (*handler)(*unsigned* index))

Description

Register a function to handle a general exception.

Arguments

- index - The exception vector number for which the handler should be called. This value cannot refer to an exception that must be handled by a memory exception handler. This value is passed to the handler when the handler is called.
- handler - A pointer to the kernel function that should be called when the specified general exception occurs.

Return Value

None.

CFI Checks

- The function `handler` should be a valid kernel function pointer.

Notes

Interrupts will be disabled when control is passed to the specified handler function. It is the handler function's reponsibility to re-enable interrupts if desired.

A.3.4 sva_register_memory_exception

Synopsis

void sva_register_memory_exception(*unsigned* index, *void* (*handler)(*unsigned* index, *void* * address))

Description

Register a function to handle a memory-related exception (e.g., a page fault).

Arguments

- index - The exception vector number for which the handler should be called. This value must refer to a memory exception vector. This value is passed to the handler when the handler is called.
- handler - A pointer to the kernel function that should be called when the specified general exception occurs.

Return Value

None.

CFI Checks

- The function `handler` should be a valid kernel function pointer.

Notes

Interrupts will be disabled when control is passed to the specified handler function. It is the handler function's responsibility to re-enable interrupts if desired.

A.4 Context Switching

A.4.1 sva_swap_integer

Synopsis

unsigned char sva_swap_integer(*stateID* newID, *stateID* * address)

Description

Save the current processor state into SVA VM internal memory and load the state identified by *newID* on to the processor. The identifier for the current processor state is saved into the memory pointed to by *address*.

Arguments

- **newID**: The identifier of the state that is to be loaded.
- **address**: The virtual memory address into which to store the state ID of the current processor state.

Return Value

- 0: The context switch failed.
- 1: The context switch succeeded.

CFI Checks

- `newID` must refer to a valid integer state.
- `address` must not point into the SVA memory.

Virtual Ghost Checks

- `address` must not point into Ghost Memory.

A.4.2 sva_load_fp

Synopsis

void sva_load_fp(*void* * buffer)

Description

Load the floating point native state previously saved into the specified buffer back on to the processor.

Arguments

- **buffer**: The first memory address of the buffer into which the floating point state was stored.

Return Value

None.

Notes

This intrinsic is deprecated.

A.4.3 sva_save_fp

Synopsis

void sva_save_fp(*void* * buffer, *bool* always)

Description

Save the floating point native state currently on the processor into the specified buffer back on to the processor.

Arguments

- **buffer**: The first memory address of the buffer into which the floating point state should be stored.
- **always**: If set to true, the floating point native state is unconditionally saved into the specified buffer. If set to false, then the floating point native state is only saved if it was modified since the last time that it was saved.

Return Value

- **0**: Floating point native state was not saved.
- **1**: Floating point native state was saved.

Notes

This intrinsic is deprecated.

A.5 Interrupted Program State

A.5.1 sva_was_privileged

Synopsis

void sva_was_privileged(*void*)

Description

Determine if the more recent Interrupt Context is for state that was executing in privileged mode.

Arguments

None.

Return Value

- **true**: The most recent Interrupt Context represents state that was running in privileged mode.
- **false**: The most recent Interrupt Context represents state that was running in non-privileged mode.

A.5.2 sva_icontext_lif

Synopsis

bool sva_icontext_lif(*void*)

Description

Determine if interrupts were enabled when the program execution represented by the most recent interrupt context was interrupted.

Arguments

None.

Return Value

- **true**: The most recent Interrupt Context represents state that was running with interrupts enabled.
- **false**: The most recent Interrupt Context represents state that was running with interrupts disabled.

A.5.3 sva_icontext_get_stackp

Synopsis

*void ** sva_icontext_get_stackp(*void*)

Description

Return the stack pointer contained within the most recent interrupt context.

Arguments

None.

Return Value

A pointer to the stack pointer within the most recent interrupt context is returned.

A.5.4 sva_icontext_load_retval

Synopsis

integer sva_icontext_load_retval(*void*)

Description

Load the return value out of an interrupt context for a program that was executing a system call.

Arguments

None.

Return Value

An integer value representing the current value to be returned by the system call is returned.

A.5.5 sva_icontext_save_retval

Synopsis

bool sva_icontext_save_retval(*integer* value)

Description

Set the return value of a system call within an interrupt context representing the state of the program that issued the system call.

Arguments

- **value**: The value that should be returned by the system call when the interrupt program state is placed back on to the processor.

Return Value

- **true**: The return value within the interrupt context was set.
- **false**: The return value within the interrupt context was not set. This could be due to the interrupt context being created by an exception or interrupt.

CFI Checks

- The most recent user-space interrupt context must be from interrupted program state created by a system call (as opposed to an interrupt or trap).

A.5.6 sva_icontext_commit

Synopsis

void sva_icontext_commit(*void*)

Description

Commit any interrupt context state that is currently residing on the processor into the interrupt context memory buffer.

Arguments

None.

Return Value

None.

A.5.7 sva_icontext_push

Synopsis

void sva_icontext_push(*(*f)()*, *integer* arg1, ...)

Description

Modify the most recently interrupted user-space state (user Interrupt Context) so that, when resumed on return from interrupt, the user-space program finds itself executing the function *f* with the specified arguments. Note that this instruction can take multiple arguments.

This instruction is used to implement asynchronous event delivery (e.g., signals on Unix systems).

Arguments

- **f**: A pointer to the function which should be called when the user-space state is resumed.
- **arg1**: The first argument that is passed to *f*.

Return Value

None.

Virtual Ghost Checks

- The function pointer `f` must point to a function that was declared as a valid function target for the currently executing integer state.

A.5.8 sva_icontext_save

Synopsis

void sva_icontext_save(*void*)

Description

Save the most recently interrupted state into SVA internal memory.

Arguments

None.

Return Value

None.

A.5.9 sva_icontext_load

Synopsis

void sva_icontext_load(*void*)

Description

Load into the most recent Interrupt Context the saved state from the most recent call to `sva_icontext_save()` made on this Interrupt Context.

Arguments

- **address**: A pointer to the virtual memory address into which the user-space interrupt context was saved.

Return Value

None.

A.5.10 sva_ialloca

Synopsis

*void ** sva_ialloca(*integer* size, *integer* alignment, *void ** initp)

Description

Perform an alloca on the stack of the interrupted program state, aligning the memory and initializing it from the memory pointed to by `initp`. The most recently Interrupt Context must represent interrupted user-space state. This intrinsic will also do validity checking to ensure that the interrupted application's stack pointer does not point into kernel memory.

Arguments

- **size**: The amount of memory, in bytes, to allocate on the stack contained within the interrupted program state.
- **alignment**: The memory will be aligned on a $2^{alignment}$ boundary.
- **initp**: If not NULL, `size` bytes of data will be copied from the memory pointed to by `initp` to the newly stack-allocated memory.

Return Value

A pointer to the allocated memory is returned.

CFI Checks

- The most recent interrupt-context represents interrupted user-space state.

Virtual Ghost Checks

- The interrupt-context is marked invalid to ensure that `sva_ialloca()` proceeds the call to `sva_icontext_push()`.
- The memory between `initp` and `initp` + `size` cannot be within SVA memory.
- The memory between `initp` and `initp` + `size` cannot be within Ghost Memory.
- The `alignment` value must be less than 64 for a 64-bit address space.

A.5.11 sva_iunwind

Synopsis

void sva_iunwind(void)

Description

Unwind control flow on the most recently created interrupted state (Interrupt Context) so that, when resumed, the interrupted state beings execution at the instruction immediately following the previous invoking instruction (e.g., *invoke*).

Arguments

None.

Return Value

None.

A.5.12 sva_init_icontext

Synopsis

uint sva_init_icontext(*void ** stackp, *uint* stacklen, *(*f)()*, *integer* arg1, ...)

Description

Initialize a new stack, interrupt context, and integer state. The integer state is initialized so that control flow resumes in the specified function which will be called with the specified arguments. The function will appear to have been called by the sc_ret function.

The interrupt context will be initialized so that it is identical to the current interrupt context with the exception that it will be using the specified kernel stack.

The current interrupt context must represent interrupted user-space state.

Arguments

- **stackp**: The first address of the kernel stack to be used for the new state.
- **stacklen**: The length, in bytes, of the new kernel stack.
- **f**: A pointer to the function which should be called when the state is loaded back on to the processor.
- **arg1**: The first argument that is passed to *f*.

Return Value

None.

CFI Checks

- The function `f` should be a pointer to a kernel function.

A.5.13 sva_reinit_icontext

Synopsis

uint sva_reinit_icontext(*void* * stackp, *bool* priv, *(*f)()*, *integer* arg1, ...)

Description

Reinitialize the most recent interrupt context so that, when it resumes on the processor, execution begins in the specified function with the specified argument. If the privilege level is changed to unprivileged, the stack pointer is also reset to the specified value.

Any secure memory mappings associated with the running integer state are discarded.

Arguments

- **stackp**: The first address of the stack to be used for the new state.
- **priv**: A flag indicating whether the interrupt context should represent privileged or unprivileged program state.
- **f**: A pointer to the function which should be called when the state is loaded back on to the processor.
- **arg1**: The first argument that is passed to *f*.

Return Value

None.

A.6 Exceptions

A.6.1 sva_invoke_memcpy

Synopsis

unsigned long sva_invoke_memcpy(*void* * dest, *void* * start, *unsigned long* length)

Description

Copy no more than *length* bytes from the buffer starting at *start* to the buffer starting at *end*. The buffers are not allowed to overlap.

If a hardware fault occurs, unwinding the state will cause execution to resume such that the `sva_invoke_memcpy()` will return the number of bytes successfully copied before the fault occurred.

Arguments

- **dest**: A pointer to the memory buffer into which to copy the data.
- **src**: A pointer to the memory buffer from which to copy the data.
- **length**: The number of bytes to copy from the source memory buffer to the destination memory buffer.

Return Value

The number of bytes successfully copied is returned.

A.7 Bitcode Translation

A.7.1 sva_translate

Synopsis

*void ** sva_translate(*void ** bitcode, *char ** name, *bool* kernelMode)

Description

Return a pointer to the function named *name* in the SVA bitcode pointed to by *bitcode*, translating to native code as necessary. If *kernelMode* is set, the code will be used while the processor is running in the privileged mode; otherwise, the code is assumed to be user-space code.

Arguments

- **bitcode**: A pointer to the first address of SVA bitcode to be used for the translation.
- **name**: The name of the function whose native code starting address should be returned.
- **kernelMode**: A flag indicating whether the native code should be used in user-mode or kernel mode.

Return Value

Zero is returned if the bitcode does not have a function with the specified name. Otherwise, a pointer to the native code translation of the specified function is returned.

A.8 Memory Management

A.8.1 sva_mm_load_pagetable

Synopsis

void sva_mm_load_pagetable(*void ** pt)

Description

Make the specified page table the active page table.

Arguments

- pt - A pointer to the top level page table page.

Return Value

None.

CFI Checks

- The pointer `pt` must be on a page boundary pointing into the direct map.
- The pointer `pt` must point to a virtual address which is mapped to a physical page that has been declared as an L4 page table page.

A.8.2 sva_mm_save_pagetable

Synopsis

*void ** sva_mm_save_pagetable(*void*)

Description

Return the pointer to the currently active top-level page table page.

Arguments

None.

Return Value

This instruction returns a pointer to the top level page table page.

A.8.3 sva_mm_flush_tlbs

Synopsis

void sva_mm_flush_tlbs(*bool* flushGlobal)

Description

Flush all TLB entries contained on the processor for translations not marked as global. If the flushGlobal operand is true, then flush page translations for global pages as well.

Arguments

- flushGlobal - If true, page translations marked as global are also flushed from the TLB.

Return Value

None.

A.8.4 sva_mm_flush_tlb

Synopsis

void sva_mm_flush_tlb(*void* * address)

Description

Flush TLB entries (both local and global) for page translations that map the specified address to a physical page location.

Arguments

- address - The virtual address for which TLB page translations should be flushed.

Return Value

None.

A.8.5 sva_mm_flush_wcache

Synopsis

void sva_mm_flush_wcache(*void*)

Description

Flush the write cache.

Arguments

None.

Return Value

None.

A.8.6 sva_declare_ptp

Synopsis

void sva_declare_ptp(*void ** ptpptr, *integer* level)

Description

Mark the page starting at the specified virtual address within the direct map as a Page Table Page. The SVA VM will modify the page protections so that the system software can no longer make direct modifications to the page and will permit the page to be used as a page table page at the specified level in the page table hierarchy.

Arguments

- ptpptr - A pointer to the first virtual address of the page that will be used as a Page Table Page.
- level - The level within the page table hierarchy at which the page table page will be used (e.g., an L2 page table page would have a level of 2).

Return Value

None.

CFI Checks

- The virtual address `ptpptr` is an address within the direct map.
- The physical page pointed to by `ptpptr` must not be mapped into any other virtual address.
- The physical page pointed to by `ptpptr` must not be used as a code page, SVA data page, or a page table page belonging to another level other than `level`.
- The physical page is zeroed to prevent stale data within the page from being used as page translations.

Virtual Ghost Checks

- The physical page pointed to by `ptpptr` must not be used as a Ghost Memory page or a Ghost Memory Page Table Page.

A.8.7 sva_release_ptp

Synopsis

void sva_release_ptp(*void* * ptpptr)

Description

Release the page starting at the specified virtual address in the direct map from the set of Page Table Pages so that the system software may use the page as regular memory again. Once released, a page cannot be used as a page table page without a subsequent call to `sva_declare_ptp()`.

Arguments

- ptpptr - A pointer to the first virtual address of the page in the direct map that will no longer be used as a Page Table Page.

Return Value

None.

CFI Checks

- The physical page pointed to by `ptpptr` must be a page table page.

- The physical page pointed to by `ptpptr` must have a reference count of zero.

A.8.8 sva_update_l1_mapping

Synopsis

void sva_update_l1_mapping(*void ** pteptr, *unsigned int* trans)

Description

Add the specified virtual-to-physical page translation into an L1 Page Table Page at the specified location.

Arguments

- pteptr - A virtual address pointing into a Level 1 Page Table Page.
- trans - An x86 page translation entry.

Return Value

None.

CFI Checks

- `pteptr` must point into the direct map.
- The virtual address represented by `pteptr` must not be within SVA memory.
- The physical page pointed to by `pteptr` must be an L1 page.
- If `trans` has the valid translation bit set, then either:
  1. The physical page in `trans` must be an unused page; or
  2. The physical page in `trans` must be a page table page with read-only permission.

Virtual Ghost Checks

- `pteptr` must point into an L1, non-ghost page table page.
- The physical page in `trans` cannot be a ghost physical page if the valid bit is set.

A.8.9 sva_update_l2_mapping

Synopsis

void sva_update_l2_mapping(*void ** pteptr, *unsigned int* trans)

Description

Add the specified virtual-to-physical page translation into an L2 Page Table Page at the specified location.

Arguments

- pteptr - A virtual address pointing into a Level 2 Page Table Page.
- trans - An x86 page translation entry.

Return Value

None.

CFI Checks

- The pointer `pteptr` must pointer into the direct map.
- The virtual address represented by `pteptr` must not be within SVA memory.
- The physical page pointed to by `pteptr` must be an L2 page.
- The physical page in `trans` must refer to an L1 page if `trans` has the valid translation bit set.

A.9 Virtual Ghost Application Instructions

A.9.1 sva_alloc_ghostmem

Synopsis

pointer sva_alloc_ghostmem(*pointer* address, *integer* size)

Description

Map `size` bytes into the virtual address specified by `address` (which must be within the ghost memory region).

Arguments

- address: The ghost virtual address at which to map memory.
- size: The number of bytes to map at the specified address.

Return Value

If the allocation cannot be performed, NULL is returned. Otherwise, a pointer to the first byte of the allocated ghost memory is returned.

Virtual Ghost Checks

- The value `address` lies within the ghost address space.

A.9.2 sva_free_ghostmem

Synopsis

void sva_free_ghostmem(*pointer* address, *integer* size)

Description

Unmap `size` bytes from the virtual address specified by `address` (which must be within the ghost memory region).

Arguments

- address: The ghost virtual address at which to unmap memory.
- size: The number of bytes to map at the specified address.

Return Value

Virtual Ghost Checks

- The value `address` lies within the ghost address space.

A.9.3 sva_validate_target

Synopsis

void sva_validate_target(*pointer* f)

Description

Specify that the function `f` can be a target of `sva_icontext_push()`.

Arguments

- f: The function pointer to validate.

Return Value

None.

A.9.4 sva_get_key

Synopsis

void sva_get_key(*pointer* keyAddr, *integer* size)

Description

Retrieve the application key from the SVA Virtual Machine.

Arguments

- keyAddr: The application address into which the key should be written.
- size: The size of the memory buffer for the key in bytes.

Return Value

None.

Virtual Ghost Checks

- The memory buffer for the key is located within the application address space or ghost address space.

Appendix B

Control-Flow Integrity Proofs

This chapter contains the Coq code that defines the semantics for the KCoFI system and proves that the semantics maintain control-flow integrity.

B.1 TLB.v

```
Require Import Arith.

Inductive ReadTy : Type := | Read | NoRead.

Inductive WriteTy : Type := | Write | NoWrite.

Inductive ExecTy : Type := | Exec | NoExec.

Inductive TLBTy : Type :=
  | emptyTLB
  | TLB : nat → ReadTy → WriteTy → ExecTy → TLBTy.

Definition definedTLB (tlb : TLBTy) :=
  match tlb with
  | emptyTLB ⇒ False
  | TLB n R W X ⇒ True
end.

Definition getPhysical (tlb : TLBTy) :=
  match tlb with
  | emptyTLB ⇒ 0
  | TLB n R W X ⇒ n
end.

Definition TLBPermitsRead (tlb : TLBTy) : Prop :=
  match tlb with
```

```
  | emptyTLB ⇒ False
  | TLB n Read W X ⇒ True
  | TLB n NoRead W X ⇒ False
end.

Definition TLBPermitsWrite (tlb : TLBTy) : Prop :=
  match tlb with
  | emptyTLB ⇒ False
  | TLB n R Write X ⇒ True
  | TLB n R NoWrite X ⇒ False
end.

Definition TLBPermitsExec (tlb : TLBTy) : Prop :=
  match tlb with
  | emptyTLB ⇒ False
  | TLB n R W Exec ⇒ True
  | TLB n R W NoExec ⇒ False
end.

Lemma PermitsWriteImpliesWrite : ∀ (n : nat) (r : ReadTy) (w : WriteTy) (e : ExecTy),
TLBPermitsWrite (TLB n r w e) → w = Write.
intros.
induction w.
reflexivity.
contradiction H.
Qed.
```

B.2 Instructions.v

```
Require Import TLB.
Require Import Arith.

Inductive tm : Type :=
  | val : nat → tm
  | sec : tm
```

| *ldi* : *nat* → *tm*

| *lda* : *nat* → *tm*

| *sta* : *nat* → *tm*

| *add* : *nat* → *tm*

| *sub* : *nat* → *tm*

| *map* : *nat* → *TLBTy* → *tm*

| *jmp* : *tm*

| *jeq* : *nat* → *tm*

| *jne* : *nat* → *tm*

| *trap* : *tm*

| *iret* : *tm*

| *svaDeclareStack* : *nat* → *nat* → *tm*

| *svaLoadPGTable* : *tm*

| *svaInitStack* : *nat* → *tm*

| *svaSwap* : *tm*

| *svaRegisterTrap* : *tm*

| *svaSaveIcontext* : *tm*

| *svaLoadIcontext* : *tm*

| *svaPushFunction* : *nat* → *tm*

| *jsr* : *tm*

| *ret* : *tm*.

B.3 Memory.v

`Require Import` *Arith*.

`Require Import` *List*.

`Require Export` *Instructions*.

`Definition` *store* := *list tm*.

`Definition` *emptyStore* := *nil* : *list tm*.

`Definition` *lookup* (*n* : *nat*) (*st* : *store*) := *nth n st* (*val* 0).

`Fixpoint replace` {*A*:`Type`} (*n*:*nat*) (*x*:*A*) (*l*:*list A*) : *list A* :=

```
  match l with
  | nil ⇒ nil
  | h :: t ⇒
    match n with
    | O ⇒ x :: t
    | S n' ⇒ h :: replace n' x t
    end
  end.

Lemma writeEmptyStore: ∀ (n : nat) (v : Instructions.tm), replace n v nil = nil.
intros.
induction n.
auto.
auto.
Qed.

Lemma readEmptyStore: ∀ (n1 n2 : nat), lookup n1 nil = lookup n2 nil.
Proof.
intros.
induction n2.
induction n1.
auto.
auto.
assert (∀ n : nat, lookup n nil = val 0).
intros.
induction n.
auto.
auto.
apply H.
Qed.

Lemma succRead: ∀ (n : nat) (v : Instructions.tm) (DS : store), lookup (S n) (v :: DS) = lookup n DS.
Proof.
intros.
```

unfold *lookup*.

auto.

Qed.

Theorem *sameRead* : ∀ (*DS* : *store*) (*n1 n2* : *nat*) (*v* : *Instructions.tm*),

(*n1* ≠ *n2*) → *lookup* (*n1*) *DS* = *lookup* (*n1*) (replace *n2 v DS*).

Proof.

intro *DS*.

induction *DS*.

intros.

rewrite → *writeEmptyStore*.

reflexivity.

intros.

destruct *n1*.

destruct *n2*.

simpl.

contradiction H.

reflexivity.

simpl.

unfold *lookup*.

simpl.

reflexivity.

destruct *n2*.

simpl.

unfold *lookup*.

simpl.

reflexivity.

rewrite → *succRead*.

simpl.

rewrite → *succRead*.

apply *IHDS*.

auto.

```
Qed.
```

B.4 MMU.v

```
Require Import Arith.

Require Import List.

Require Export Memory.

Require Export TLB.

Definition MMU := list TLBTy.

Definition emptyMMU := nil : list TLBTy.

Definition getTLB (n : nat) (mmu : MMU) := nth n mmu emptyTLB.

Fixpoint updateTLB (virt:nat) (tlb:TLBTy) (mmu:MMU) : MMU :=
  match mmu with
  | nil ⇒ nil
  | h :: t ⇒
    match virt with
    | O ⇒ tlb :: t
    | S n' ⇒ h :: updateTLB n' tlb t
    end
  end.

Definition MMUSet := list MMU.

Definition emptyMMUSet := nil : MMUSet.

Definition getMMU (asid : nat) (mmu : MMUSet) := nth asid mmu emptyMMU.

Fixpoint updateMMU (virt:nat) (asid : nat) (tlb:TLBTy) (mmus:MMUSet) : MMUSet :=
  match mmus with
  | nil ⇒ nil
  | h :: t ⇒
    match asid with
    | O ⇒ (updateTLB virt tlb h) :: t
    | S n' ⇒ h :: updateMMU virt n' tlb t
    end
```

```
  end.

Definition canRead (va : nat) (mmu : MMU) : Prop := TLBPermitsRead (getTLB va mmu).

Definition canWrite (va : nat) (mmu : MMU) : Prop := TLBPermitsWrite (getTLB va mmu).

Definition canExec (va : nat) (mmu : MMU) : Prop := TLBPermitsExec (getTLB va mmu).

Definition vLookup (va : nat) (mmu : MMU) (st : store) :=
  lookup (getPhysical (getTLB va mmu)) st.

Definition vaLookup (va : nat) (asid : nat) (mmu : MMU) (st : store) :=
  lookup (getPhysical (getTLB va mmu)) st.

Lemma noTLBinMMU : ∀ (v : nat),
(getTLB v nil) = emptyTLB.
Proof.
intros.
induction v.
auto.
unfold getTLB.
simpl.
auto.
Qed.

Lemma updateEmptyMMU: ∀ (v : nat) (tlb : TLBTy), updateTLB v tlb nil = nil.
intros.
induction v.
auto.
auto.
Qed.

Lemma oneOrAnother : ∀ (v : nat) (tlb a : TLBTy) (mmu : MMU),
(updateTLB v tlb (a :: mmu) = tlb :: mmu) ∨
(updateTLB v tlb (a :: mmu) = a :: (updateTLB (pred v) tlb mmu)).
Proof.
intros.
destruct v.
```

left.

auto.

right.

simpl.

auto.

Qed.

Lemma *skipTLB*: ∀ (*n* : *nat*) (*tlb* : *TLBTy*) (*mmu* : *MMU*),

(*getTLB* (*S n*) (*tlb* :: *mmu*)) = (*getTLB n* (*mmu*)).

Proof.

intros.

unfold *getTLB*.

simpl.

reflexivity.

Qed.

Theorem *sameMMULookup* : ∀ (*mmu* : *MMU*) (*n m* : *nat*) (*tlb* : *TLBTy*),

n ≠ *m* → *getTLB* (*n*) (*mmu*) = *getTLB* (*n*) (*updateTLB m tlb mmu*).

Proof.

intro.

induction *mmu*.

intros.

rewrite → *updateEmptyMMU*.

auto.

intros.

destruct *n*.

destruct *m*.

contradiction H.

auto.

simpl.

unfold *getTLB*.

simpl.

```
auto.
destruct m.
unfold getTLB.
simpl.
auto.
rewrite → skipTLB.
simpl.
rewrite → skipTLB.
apply IHmmu.
auto.
Qed.
Theorem sameMMURead : ∀ (n m : nat) (tlb : TLBTy) (mmu : MMU) (ds : store),
  n ≠ m → vLookup (n) mmu ds = vLookup (n) (updateTLB m tlb mmu) ds.
Proof.
intros.
unfold vLookup.
rewrite ← sameMMULookup.
auto.
apply H.
Qed.
Theorem diffTLBImpliesDiffVAs : ∀ (v1 v2 : nat) (mmu : MMU),
((getTLB v1 mmu) ≠ (getTLB v2 mmu)) → v1 ≠ v2.
Proof.
intros.
contradict H.
rewrite → H.
reflexivity.
Qed.
Theorem sameMMUPerms : ∀ (n m : nat) (tlb : TLBTy) (mmu : MMU),
  n ≠ m → canWrite n mmu = canWrite n (updateTLB m tlb mmu).
Proof.
```

```
intros.
induction mmu.
unfold updateTLB.
simpl.
rewrite → updateEmptyMMU.
auto.
destruct n.
destruct m.
contradiction H.
reflexivity.
simpl.
unfold canWrite.
unfold getTLB.
simpl.
auto.
destruct m.
simpl.
unfold canWrite.
unfold getTLB.
simpl.
auto.
unfold canWrite.
simpl.
rewrite → skipTLB.
rewrite → skipTLB.
assert (n ≠ m).
contradict H.
auto.
rewrite ← sameMMULookup.
auto.
auto.
```

```
Qed.

Theorem tlbSet : ∀ (v : nat) (tlb : TLBTy) (mmu : MMU),
definedTLB (getTLB v mmu) → getTLB v (updateTLB v tlb mmu) = tlb.
Proof.
intros.
generalize dependent tlb.
generalize dependent v.
induction mmu.
intros.
rewrite → noTLBinMMU in H.
contradiction H.
intros.
destruct v.
auto.
simpl.
unfold getTLB.
simpl.
unfold getTLB in IHmmu.
apply IHmmu.
apply H.
Qed.

Lemma updateEmptyMMUS: ∀ (v asid : nat) (tlb : TLBTy), updateMMU v asid tlb nil = nil.
intros.
induction asid.
auto.
auto.
Qed.

Lemma noMMUinMMUS : ∀ (asid : nat),
(getMMU asid nil) = emptyMMU.
Proof.
intros.
```

induction *asid*.

auto.

unfold *getMMU*.

simpl.

auto.

Qed.

Lemma *skipMMU*: ∀ (*asid* : *nat*) (*mmu* : *MMU*) (*mmus* : *MMUSet*),

(*getMMU* (*S asid*) (*mmu* :: *mmus*)) = (*getMMU asid mmus*).

Proof.

intros.

unfold *getTLB*.

simpl.

reflexivity.

Qed.

Theorem *sameMMUSet* : ∀ (*mmus* : *MMUSet*) (*v asid* : *nat*) (*tlb* : *TLBTy*),

updateTLB v tlb (*getMMU asid mmus*) = *getMMU asid* (*updateMMU v asid tlb mmus*).

Proof.

intro.

induction *mmus*.

intros.

rewrite → *updateEmptyMMUS*.

rewrite → *noMMUinMMUS*.

rewrite → *updateEmptyMMU*.

auto.

intros.

destruct *asid*.

auto.

rewrite → *skipMMU*.

simpl.

rewrite → *skipMMU*.

```
apply IHmmus.
Qed.

Theorem sameVALookup : ∀ (mmus : MMUSet) (v v0 asid : nat) (tlb : TLBTy),
v ≠ v0 → (getTLB v (getMMU asid mmus)) = getTLB v (getMMU asid (updateMMU v0 asid tlb mmus)).
Proof.
intro.
induction mmus.
intros.
rewrite → updateEmptyMMUS.
auto.
intros.
destruct asid.
unfold getMMU.
simpl.
apply sameMMULookup.
auto.
rewrite → skipMMU.
simpl.
rewrite → skipMMU.
apply IHmmus.
auto.
Qed.
```

B.5 Stack.v

```
Require Import Arith.
Require Import List.
Inductive Stack : Type :=
  stack : nat → nat → Stack.
Notation "( x , y )" := (stack x y).
```

```
Definition stackWellFormed (s : Stack) : Prop :=
  match s with
    (x,y) ⇒ (0 < x < y)
  end.

Fixpoint stacksWellFormed (sl : list Stack) : Prop :=
  match sl with
  | nil ⇒ True
  | h :: t ⇒ (stackWellFormed h) ∧ stacksWellFormed (t)
  end.

Definition within (n : nat) (s : Stack) : Prop :=
  match s with
    (x,y) ⇒ (x ≤ n ≤ y)
  end.

Fixpoint Within (n : nat) (sl : list Stack) : Prop :=
  match sl with
  | nil ⇒ False
  | h :: t ⇒ (within n h) ∨ (Within n t)
  end.
```

B.6 IC.v

```
Require Import Arith.
Require Import List.

Inductive InterruptContext : Type :=
  | IC : nat →
         nat →
         nat →
         nat →
         InterruptContext.

Definition ICStack := list InterruptContext.
Definition emptyICStack := nil : list InterruptContext.
```

```
Definition itop (ics : ICStack) : InterruptContext := nth 0 ics (IC 0 0 0 0).

Definition push (ic : InterruptContext) (ics : ICStack) := (ic :: ics).

Definition pop (ics : ICStack) :=
  match ics with
  | nil ⇒ nil
  | h :: t ⇒ t
end.

Definition getICPC (ic : InterruptContext) :=
  match ic with
  | IC Reg PC SP Priv ⇒ PC
end.
```

B.7 Thread.v

```
Require Import Arith.

Require Import Bool.

Require Import List.

Require Export IC.

Inductive SVAThread : Type :=
  | Thread : bool →
             nat →
             ICStack →
             ICStack →
             SVAThread.

Definition emptyThread := Thread false 0 nil nil.

Definition ThreadList := list SVAThread.

Definition emptyTL := nil : list SVAThread.

Definition getThread (id : nat) (t : ThreadList) := nth id t emptyThread.

Fixpoint updateThread (id:nat) (thread:SVAThread) (tl:ThreadList) : ThreadList :=
  match tl with
  | nil ⇒ nil
```

```
  | h :: t ⇒
    match id with
    | O ⇒ thread :: t
    | S n' ⇒ h :: updateThread n' thread t
    end
  end.

Definition getThreadPC (t : SVAThread) :=
  match t with
  | Thread valid PC stack istack ⇒ PC
end.

Definition canThreadSwap (t : SVAThread) :=
  match t with
  | Thread canSwap PC stack istack ⇒ canSwap
end.

Definition getThreadICL (t : SVAThread) :=
  match t with
  | Thread canSwap PC stack istack ⇒ stack
  end.

Definition getThreadSICL (t : SVAThread) :=
  match t with
  | Thread canSwap PC stack istack ⇒ istack
  end.

Definition getThreadICList (id : nat) (tl : ThreadList) :=
  getThreadICL (getThread id tl).

Definition getThreadSICList (id : nat) (tl : ThreadList) :=
  getThreadSICL (getThread id tl).

Definition threadOnCPU (newpc : nat) (t : SVAThread) :=
  match t with
  | Thread canSwap PC stack istack ⇒ Thread true newpc stack istack
end.
```

```
Definition threadOffCPU (t : SVAThread) :=
  match t with
  | Thread canSwap PC stack istack ⇒ Thread false 0 stack istack
end.

Definition pushSIC (tid : nat) (tl : ThreadList) :=
  updateThread tid
               (Thread
                  false
                  (getThreadPC (getThread tid tl))
                  (getThreadICList tid tl)
                  (push
                     (itop (getThreadICList tid tl))
                     (getThreadSICList tid tl)))
               tl.

Definition popSIC (tid : nat) (tl : ThreadList) :=
  updateThread tid
               (Thread
                  false
                  (getThreadPC (getThread tid tl))
                  (push
                     (itop (getThreadSICList tid tl))
                     (pop (getThreadICList tid tl)))
                  (pop (getThreadSICList tid tl)))
               tl.

Definition pushIC (Reg PC SP: nat) (tid : nat) (tl : ThreadList) :=
  updateThread tid
               (Thread
                  false
                  (getThreadPC (getThread tid tl))
                  (push (IC Reg PC SP 0) (pop (getThreadICList tid tl)))
                  (getThreadSICList tid tl))
```

tl.

Theorem *updateMaintainsListLength*: ∀ (*tid* : *nat*) (*t* :*SVAThread*) (*tl* :*ThreadList*),

length tl = *length* (*updateThread tid t tl*).

Proof.

intros.

generalize dependent *tid*.

induction *tl*.

intros.

simpl.

unfold *updateThread*.

destruct *tid*.

auto.

auto.

intros.

simpl.

destruct *tid*.

unfold *updateThread*.

simpl.

auto.

unfold *updateThread*.

simpl.

auto.

Qed.

B.8 Config.v

Require Import *Arith*.

Require Import *List*.

Require Export *Stack*.

Require Export *MMU*.

Require Export *IC*.

```
Require Export Thread.

Inductive config : Type :=
  | C : MMU →
        store →
        tm →
        nat →
        nat →
        list nat →
        nat →
        nat →
        list Stack →
        nat →
        nat →
        nat →
        ThreadList →
        nat →
        nat →
        list nat →
        nat →
        config.

Definition getReg (c : config) : tm :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ Reg
end.

Definition getPC (c : config) : nat :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ PC
end.

Definition getCFG (c : config) : list nat :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ CFG
```

```
end.

Definition getTH (c : config) : nat :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ th
end.

Definition getTextStart (c : config) : nat :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ cs
end.

Definition getTextEnd (c : config) : nat :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ ce
end.

Definition getStore (c : config) : store :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ DS
end.

Definition getCMMU (c : config) : MMU :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ MMU
end.

Definition getGhostStart (c : config) : nat :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ gvs
end.

Definition getGhostEnd (c : config) : nat :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ gve
end.

Definition getGhost (c : config) : list nat :=
```

```
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ gl
end.

Definition canExecConfig (c : config) : Prop :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒
      (canExec PC MMU)
end.

Definition incPC (c : config) : config :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒
    C MMU DS Reg (S PC) SP CFG cs ce st asid tid ntid tl gvs gve gl th
end.

Definition setReg (c : config) (n : nat) : config :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒
    C MMU DS (val n) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th
end.

Definition vlookup (vaddr : nat) (c : config) :=
  (vLookup vaddr (getCMMU c) (getStore c)).

Definition getInsn (c : config) : tm := (vlookup (getPC c) c).

Definition getCurrThread (c : config) :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ tid
end.

Definition getThreadList (c : config) : ThreadList :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒ tl
end.

Definition setThreadList (ntl : ThreadList) (c : config) : config :=
```

```
  match c with
  | (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) ⇒
    (C MMU DS Reg PC SP CFG cs ce st asid tid ntid ntl gvs gve gl th)
end.

Definition validConfig (c : config) : Prop :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒


      (0 < cs ≤ ce) ∧ (0 < gvs ≤ gve) ∧ (stacksWellFormed st)
end.

Definition validThreadIDs (c : config) : Prop :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒
      (tid < length tl) ∧ (ntid < length tl)
end.

Definition validThread (t : SVAThread)
                       (cfg : list nat)
                       (mmu : MMU)
                       (ds : store) : Prop :=
  match t with
  | Thread false pc icl isl ⇒ True
  | Thread true pc icl isl ⇒
      (In pc cfg) ∨ ((vLookup (minus pc 1) mmu ds) = svaSwap)
  end.

Definition validCFG3 (cfg : list nat) (mmu : MMU) (cs ce : nat) :=
  ∀ (f : nat), (In f cfg) →
    (cs ≤ (getPhysical (getTLB f mmu)) ≤ ce) ∧
    (cs ≤ (getPhysical (getTLB (pred f) mmu)) ≤ ce).

Fixpoint validCFG2 (cfg : list nat) (mmu : MMU) (cs ce : nat) : Prop :=
  match cfg with
  | nil ⇒ True
```

```
  | n :: t ⇒
    cs ≤ (getPhysical (getTLB n mmu)) ≤ ce ∧
    cs ≤ (getPhysical (getTLB (pred n) mmu)) ≤ ce ∧
    (validCFG2 t mmu cs ce)
  end.

Definition validCFG (c : config) (cfg : list nat) : Prop :=
  validCFG2 cfg (getCMMU c) (getTextStart c) (getTextEnd c).

Lemma cfg23 : ∀ (cfg : list nat) (mmu : MMU) (cs ce : nat),
  validCFG2 cfg mmu cs ce → validCFG3 cfg mmu cs ce.
Proof.
intros cfg mmu.
induction cfg.

intros.
unfold validCFG3.
intros.
contradiction.

intros.
unfold validCFG3.
intros.
unfold validCFG2 in H.
destruct H as [H1 H2].
destruct H2 as [H2 H3].
fold validCFG2 in H3.
unfold In in H0.
destruct H0 as [I1 | I2].
rewrite ← I1.
auto.
apply IHcfg.
auto.
auto.
Qed.
```

```
Lemma cfg32 : ∀ (cfg : list nat) (mmu : MMU) (cs ce : nat),
  validCFG3 cfg mmu cs ce → validCFG2 cfg mmu cs ce.
Proof.
intros cfg mmu.
induction cfg.
intros.
unfold validCFG2.
auto.
intros.
unfold validCFG2.
split.
apply H.
unfold In.
auto.
split.
apply H.
unfold In.
auto.
fold validCFG2.
apply IHcfg.
unfold validCFG3 in H.
unfold validCFG3.
intros.
apply H.
apply in_cons.
auto.
Qed.
Fixpoint validThreadList (tl : ThreadList)
                         (cfg : list nat)
                         (mmu : MMU)
                         (ds : store) : Prop :=
```

```
  match tl with
  | nil ⇒ True
  | h :: t ⇒ (validThread h cfg mmu ds) ∧ (validThreadList t cfg mmu ds)
  end.

Definition threadInText (t : SVAThread) (cfg : list nat) (mmu : MMU) (cs ce : nat) : Prop :=
  match t with
  | Thread false pc icl isl ⇒ True
  | Thread true pc icl isl ⇒
    cs ≤ (getPhysical (getTLB pc mmu)) ≤ ce ∧
    ((In pc cfg) ∨ (cs ≤ (getPhysical (getTLB (pred pc) mmu)) ≤ ce))
  end.

Fixpoint threadListInText (tl : ThreadList) (cfg : list nat) (mmu : MMU) (cs ce : nat) : Prop :=
  match tl with
  | nil ⇒ True
  | h :: t ⇒ (threadInText h cfg mmu cs ce) ∧ (threadListInText t cfg mmu cs ce)
  end.

Definition textNotWriteable (c : config) : Prop :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒


      ∀ (v : nat),
      (cs ≤ (getPhysical (getTLB v MMU)) ≤ ce) → not (canWrite v MMU)
end.

Definition textMappedOnce (c : config) : Prop :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒
      ∀ (v1 v2 : nat),
      (cs ≤ (getPhysical (getTLB v1 MMU)) ≤ ce) ∧ (v1 ≠ v2) →
      (getPhysical (getTLB v1 MMU)) ≠ (getPhysical (getTLB v2 MMU))
end.

Definition textMappedLinear (c : config) : Prop :=
```

```
  ∀ (v : nat),
  ((getTextStart c) ≤ (getPhysical (getTLB v (getCMMU c))) ≤ (getTextEnd c))
  →
  ((getTextStart c) ≤ (getPhysical (getTLB (S v) (getCMMU c))) ≤ (getTextEnd c))
  ∨ ((vlookup v c) = jmp).

Definition pcInText (c : config) : Prop :=
  ((getTextStart c) ≤ (getPhysical (getTLB (getPC c) (getCMMU c))) ≤ (getTextEnd c)).

Definition makeIC (c : config) : InterruptContext :=
  match c with
  | C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th ⇒
    match Reg with
    | (val n) ⇒ (IC n PC SP 0)
    | _ ⇒ (IC 0 PC SP 0)
  end
end.
```

B.9 Semantics.v

```
Require Import Arith.
Require Import List.
Require Import Coq.Logic.Classical_Prop.
Require Export Config.
Reserved Notation " t '==>' t' " (at level 40).
Inductive step : config → config → Prop :=
  | ST_LoadImm : ∀ c n,
      (canExecConfig c) ∧ ((getInsn c) = ldi n) →
      c ==> (setReg (incPC c) n)
  | ST_Load : ∀ MMU DS Reg PC SP CFG n cs ce st asid tid ntid tl
                gvs gve gl th t,
      (canExec PC MMU) ∧ (vaLookup PC asid MMU DS) = lda n ∧
      (canRead n MMU) ∧ (vaLookup n asid MMU DS = t) ∧
```

$((n < gvs) \vee (n > gve)) \rightarrow$

$(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

$==>$

$(C\ MMU\ DS\ t\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

| $ST\_Store : \forall\ MMU\ DS\ Reg\ PC\ SP\ CFG\ n\ cs\ ce\ st\ asid\ tid\ ntid\ tl$

$gvs\ gve\ gl\ th,$

$(canExec\ PC\ MMU) \wedge (vaLookup\ PC\ asid\ MMU\ DS) = sta\ n \wedge$

$(canWrite\ n\ MMU) \wedge$

$((n < gvs) \vee (n > gve)) \rightarrow$

$(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) ==>$

$(C\ MMU\ (\texttt{replace}\ (getPhysical\ (getTLB\ n\ MMU))\ Reg\ DS)\ Reg\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

| $ST\_Add : \forall\ MMU\ DS\ PC\ SP\ CFG\ n\ cs\ ce\ st\ asid\ tid\ ntid\ tl$

$gvs\ gve\ gl\ th\ v,$

$(canExec\ PC\ MMU) \wedge (vaLookup\ PC\ asid\ MMU\ DS) = add\ n \rightarrow$

$(C\ MMU\ DS\ (val\ v)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

$==>$

$(C\ MMU\ DS\ (val\ (v + n))\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

| $ST\_Sub : \forall\ MMU\ DS\ PC\ SP\ CFG\ n\ cs\ ce\ st\ asid\ tid\ ntid\ tl$

$gvs\ gve\ gl\ th\ v,$

$(canExec\ PC\ MMU) \wedge (vaLookup\ PC\ asid\ MMU\ DS) = sub\ n \rightarrow$

$(C\ MMU\ DS\ (val\ v)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

$==>$

$(C\ MMU\ DS\ (val\ (v\ -\ n))\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

| $ST\_Jmp : \forall\ MMU\ DS\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl$

$gvs\ gve\ gl\ th\ v,$

$(canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = jmp) \wedge (In\ v\ CFG) \rightarrow$

$(C\ MMU\ DS\ (val\ v)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

$==>$

$(C\ MMU\ DS\ (val\ v)\ v\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$

| $ST\_Map : \forall\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ tlb$

$$
\begin{array}{l}
\qquad\qquad\quad v\ p\ gvs\ gve\ gl\ th, \\
(canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = map\ v\ tlb) \wedge \\
(p = getPhysical\ (getTLB\ v\ MMU)) \wedge \\
(((getPhysical\ tlb) < cs) \vee (ce < (getPhysical\ tlb))) \wedge \\
((p < cs) \vee (ce < p)) \wedge \\
((v < gvs) \vee (gve < v)) \wedge \\
(not\ (In\ (getPhysical\ tlb)\ gl)) \wedge \\
(definedTLB\ (getTLB\ v\ MMU)) \\
\rightarrow \\
(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
==> \\
(C\ (updateTLB\ v\ tlb\ MMU)\ DS\ Reg\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
| \ ST\_DeclareStack : \forall\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl \\
\qquad\qquad\quad s\ e\ ps\ pe\ gvs\ gve\ gl\ th, \\
(canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = svaDeclareStack\ s\ e) \wedge \\
(ps = getPhysical\ (getTLB\ s\ MMU)) \wedge \\
(pe = getPhysical\ (getTLB\ e\ MMU)) \wedge \\
((ps < cs) \vee (ce < ps)) \wedge \\
((pe < cs) \vee (ce < pe)) \wedge \\
(0 < s < e) \\
\rightarrow \\
(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
==> \\
(C\ MMU\ DS\ Reg\ (S\ PC)\ SP\ CFG\ cs\ ce\ ((s,e) ::st)\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
| \ ST\_Jeq1 : \forall\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ n \\
\qquad\qquad\quad gvs\ gve\ gl\ th, \\
(canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = jeq\ n) \wedge (In\ n\ CFG) \wedge \\
(Reg = (val\ 0)) \rightarrow \\
(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
==> \\
(C\ MMU\ DS\ Reg\ n\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)
\end{array}
$$

$$
\begin{aligned}
&| \ ST\_Jeq2 : \forall\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ n \\
&\qquad\qquad gvs\ gve\ gl\ th, \\
&\quad (canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = jeq\ n) \wedge \\
&\quad (Reg \neq (val\ 0)) \rightarrow \\
&\quad (C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
&\quad ==> \\
&\quad (C\ MMU\ DS\ Reg\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
&| \ ST\_Jne1 : \forall\ MMU\ DS\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ n \\
&\qquad\qquad gvs\ gve\ gl\ th\ vr, \\
&\quad (canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = jne\ n) \wedge (In\ n\ CFG) \wedge \\
&\quad (vr < 0) \rightarrow \\
&\quad (C\ MMU\ DS\ (val\ vr)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
&\quad ==> \\
&\quad (C\ MMU\ DS\ (val\ vr)\ n\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
&| \ ST\_Jne2 : \forall\ MMU\ DS\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ n \\
&\qquad\qquad gvs\ gve\ gl\ th\ vr, \\
&\quad (canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = jne\ n) \wedge \\
&\quad (not\ (vr < 0)) \rightarrow \\
&\quad (C\ MMU\ DS\ (val\ vr)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
&\quad ==> \\
&\quad (C\ MMU\ DS\ (val\ vr)\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
&| \ ST\_LoadPGTable: \forall\ MMU\ DS\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl \\
&\qquad\qquad gvs\ gve\ gl\ th\ vr, \\
&\quad (canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = svaLoadPGTable) \rightarrow \\
&\quad (C\ MMU\ DS\ (val\ vr)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) ==> \\
&\quad (C\ MMU\ DS\ (val\ vr)\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ vr\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) \\
&| \ ST\_InitStack : \forall\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ f \\
&\qquad\qquad gvs\ gve\ gl\ th, \\
&\quad (canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = svaInitStack\ f) \wedge \\
&\quad (In\ f\ CFG) \wedge \\
&\quad (S\ ntid < length\ tl) \wedge
\end{aligned}
$$

$(0 < length\ (getThreadICList\ tid\ tl)) \rightarrow$

$(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) ==>$

$(C\ MMU\ DS\ (val\ ntid)\ (S\ PC)\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ (S\ ntid)$

$(updateThread\ ntid\ (Thread\ true\ f\ ((itop\ (getThreadICList\ tid\ tl)) :: nil)\ nil)\ tl)\ gvs\ gve\ gl\ th)$

$|\ ST\_Swap : \forall\ MMU\ DS\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl$

$gvs\ gve\ gl\ th\ vr,$

$(canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = svaSwap) \wedge$

$((canThreadSwap\ (getThread\ vr\ tl)) = true) \wedge$

$(vr < length\ tl)$

$\rightarrow$

$(C\ MMU\ DS\ (val\ vr)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) ==>$

$(C\ MMU\ DS\ (val\ tid)\ (getThreadPC\ (getThread\ vr\ tl))\ SP\ CFG\ cs\ ce\ st\ asid\ vr\ ntid$

$(updateThread\ tid\ (threadOnCPU\ (S\ PC)\ (getThread\ tid\ tl))$

$(updateThread\ vr\ (threadOffCPU\ (getThread\ vr\ tl))\ tl))\ gvs\ gve\ gl\ th)$

$|\ ST\_Trap : \forall\ MMU\ DS\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl$

$gvs\ gve\ gl\ th\ r,$

$(canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = trap) \rightarrow$

$(C\ MMU\ DS\ (val\ r)\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) ==>$

$(C\ MMU\ DS\ (val\ r)\ th\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid$

$(updateThread\ tid$

$(Thread\ false\ 0\ (push\ (IC\ r\ (S\ PC)\ SP\ 0)\ (getThreadICList\ tid\ tl))\ (getThreadSICList\ tid\ tl))$

$tl)$

$gvs\ gve\ gl\ th)$

$|\ ST\_IRet : \forall\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl$

$gvs\ gve\ gl\ th\ popPC,$

$(canExec\ PC\ MMU) \wedge ((vaLookup\ PC\ asid\ MMU\ DS) = iret) \wedge$

$(popPC = (getICPC\ (itop\ (getThreadICList\ tid\ tl)))) \wedge$

$(0 < length\ (getThreadICList\ tid\ tl))$

$\rightarrow$

$(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) ==>$

$(C\ MMU\ DS\ Reg\ popPC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid$

```
        (updateThread tid
          (Thread false 0 (pop (getThreadICList tid tl)) (getThreadSICList tid tl)) tl)
        gvs gve gl th)
  | ST_SVARegTrap : ∀ MMU DS PC SP CFG cs ce st asid tid ntid tl
                      gvs gve gl th vr,
      (canExec PC MMU) ∧ ((vaLookup PC asid MMU DS) = svaRegisterTrap) ∧
      (In vr CFG) →
      (C MMU DS (val vr) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) ==>
      (C MMU DS (val vr) (S PC) SP CFG cs ce st asid tid ntid tl gvs gve gl vr)
  | ST_svaSaveIC : ∀ MMU DS Reg PC SP CFG cs ce st asid tid ntid tl
                      gvs gve gl th,
      (canExec PC MMU) ∧ ((vaLookup PC asid MMU DS) = svaSaveIcontext) ∧
      (0 < length (getThreadICList tid tl)) →
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) ==>
      (C MMU DS Reg (S PC) SP CFG cs ce st asid tid ntid (pushSIC tid tl) gvs gve gl th)
  | ST_svaLoadIC : ∀ MMU DS Reg PC SP CFG cs ce st asid tid ntid tl
                      gvs gve gl th,
      (canExec PC MMU) ∧
      ((vaLookup PC asid MMU DS) = svaLoadIcontext) ∧
      (0 < length (getThreadSICList tid tl)) →
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) ==>
      (C MMU DS Reg (S PC) SP CFG cs ce st asid tid ntid (popSIC tid tl) gvs gve gl th)
  | ST_svaIPushF : ∀ MMU DS PC SP CFG cs ce st asid tid ntid tl
                      gvs gve gl th f a,
      (canExec PC MMU) ∧ ((vaLookup PC asid MMU DS) = svaPushFunction a) ∧
      (In f CFG) →
      (C MMU DS (val f) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) ==>
      (C MMU DS (val f) (S PC) SP CFG cs ce st asid tid ntid (pushIC a f SP tid tl) gvs gve gl th)
  where " t '==>' t' " := (step t t').

Theorem alwaysValid : ∀ (c1 c2 : config),
(validConfig c1) ∧ (c1 ==> c2) → validConfig c2.
```

```
Proof.
intros.
destruct H as [H0 H1].
destruct H1.
destruct H.
destruct c.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
destruct H.
destruct H1.
destruct H2.
destruct H3.
destruct H4.
destruct H5.
unfold validConfig.
split.
apply H0.
split.
apply H0.
unfold stacksWellFormed.
simpl.
split.
auto.
simpl.
apply H0.
```

```
auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

Qed.

Theorem neverUnmapText : ∀ (v : nat) (c1 c2 : config),

  (c1 ==> c2) ∧

  (getTextStart c1) ≤ (getPhysical (getTLB v (getCMMU c1))) ≤ (getTextEnd c1)

  →

  getPhysical (getTLB v (getCMMU c1)) = getPhysical (getTLB v (getCMMU c2)).

Proof.

  intros.

  destruct H as [H1 H2].

  destruct H1.

  destruct c.

  auto.

  auto.

  auto.

  auto.
```

```
auto.

auto.

destruct H as [H1 H].

destruct H as [H3 H].

destruct H as [H4 H].

destruct H as [H5 H].

destruct H as [H H6].

simpl in H2.

simpl.

rewrite → H4 in H.

destruct H.

destruct H5.

assert (getTLB v0 MMU ≠ getTLB v MMU).

contradict H.

apply le_not_lt.

rewrite → H.

apply H2.

assert (v ≠ v0).

contradict H5.

rewrite → H5.

reflexivity.

assert ((getTLB v MMU) = (getTLB v (updateTLB v0 tlb MMU))).

apply sameMMULookup.

apply H7.

rewrite → H8.

auto.

assert (getTLB v0 MMU ≠ getTLB v MMU).

contradict H.

apply le_not_lt.

rewrite → H.

apply H2.
```

assert $(v \neq v0)$.

contradict H5.

rewrite $\rightarrow$ *H5*.

reflexivity.

assert $((getTLB\ v\ MMU) = (getTLB\ v\ (updateTLB\ v0\ tlb\ MMU)))$.

apply *sameMMULookup*.

apply *H7*.

rewrite $\rightarrow$ *H8*.

auto.

assert $(getTLB\ v0\ MMU \neq getTLB\ v\ MMU)$.

contradict H.

apply *le_not_lt*.

rewrite $\rightarrow$ *H*.

apply *H2*.

assert $(v \neq v0)$.

destruct *H5*.

contradict H0.

rewrite $\rightarrow$ *H0*.

reflexivity.

contradict H0.

rewrite $\rightarrow$ *H0*.

reflexivity.

assert $((getTLB\ v\ MMU) = (getTLB\ v\ (updateTLB\ v0\ tlb\ MMU)))$.

apply *sameMMULookup*.

apply *H7*.

rewrite $\rightarrow$ *H8*.

auto.

auto.

auto.

auto.

```
  auto.
  auto.
  auto.
  auto.
  auto.
  auto.
  auto.
  auto.
  auto.
  auto.
Qed.
Theorem neverWriteText : ∀ (c1 c2 : config),
(textNotWriteable c1) ∧ (c1 ==> c2) → textNotWriteable c2.
Proof.
intros.
destruct H as [H0 H1].
destruct H1.
destruct c.
auto.
auto.
auto.
auto.
auto.
auto.
destruct H.
destruct H1.
destruct H2.
```

```
destruct H3.
destruct H4.
rewrite → H2 in H4.
simpl.
intros.
unfold textNotWriteable in H0.
assert (getPhysical (getTLB v0 (updateTLB v tlb MMU)) ≠ getPhysical tlb).
contradict H3.
apply and_not_or.
split.
apply le_not_lt.
rewrite ← H3.
apply H6.
apply le_not_lt.
rewrite ← H3.
apply H6.
assert (getPhysical (getTLB v0 (updateTLB v tlb MMU)) ≠ getPhysical tlb → v0 ≠ v).
intro.
contradict H7.
rewrite → H7.
rewrite → tlbSet.
auto.
apply H5.
assert (v0 ≠ v).
apply H8.
apply H7.
rewrite ← sameMMULookup in H6.
assert (not (canWrite v0 MMU)).
apply H0.
apply H6.
rewrite ← sameMMUPerms.
```

apply *H10*.

apply *H9*.

apply *H9*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

Qed.

Theorem *neverMapTextTwice*: $\forall$ (*c1 c2* : *config*),

(*textMappedOnce c1*) $\wedge$ (*c1* $==>$ *c2*) $\rightarrow$ *textMappedOnce c2*.

Proof.

intros.

destruct *H* as [*H1 H2*].

destruct *H2*.

destruct *c*.

auto.

auto.

auto.

```
auto.

auto.

auto.

destruct H as [H2 H].

destruct H as [H3 H].

destruct H as [H4 H].

destruct H as [H5 H].

destruct H as [H6 H].

rewrite → H4 in H6.

simpl.

intros.

destruct H0 as [H7 H0].

simpl in H1.

assert (getTLB v1 MMU = (getTLB v1 (updateTLB v tlb MMU))).

apply sameMMULookup.

destruct H5.

destruct H6.

destruct H7.

contradict H7.

apply lt_not_le.

rewrite → H7.

assert (getTLB v (updateTLB v tlb MMU) = tlb).

apply tlbSet.

apply H.

rewrite → H9.

apply H5.

destruct H7.

contradict H7.

apply lt_not_le.

rewrite → H7.
```

assert $(getTLB\ v\ (updateTLB\ v\ tlb\ MMU) = tlb)$.

apply *tlbSet*.

apply *H*.

rewrite $\rightarrow$ *H9*.

apply *H5*.

destruct *H7*.

contradict H8.

apply *lt_not_le*.

rewrite $\rightarrow$ *H8*.

assert $(getTLB\ v\ (updateTLB\ v\ tlb\ MMU) = tlb)$.

apply *tlbSet*.

apply *H*.

rewrite $\rightarrow$ *H9*.

apply *H5*.

assert $(getPhysical\ (getTLB\ v1\ MMU) \neq getPhysical\ (getTLB\ v2\ MMU))$.

apply *H1*.

split.

rewrite $\leftarrow$ *H8* in *H7*.

auto.

auto.

assert $((v2 = v) \vee (v2 \neq v))$.

apply *classic*.

assert $(getTLB\ v\ (updateTLB\ v\ tlb\ MMU) = tlb)$.

apply *tlbSet*.

apply *H*.

destruct *H10*.

rewrite $\rightarrow$ *H10*.

rewrite $\rightarrow$ *H11*.

destruct *H7*.

destruct *H5*.

apply *not_eq_sym*.

contradict H5.

apply *le_not_lt*.

rewrite → *H5*.

auto.

contradict H5.

apply *le_not_lt*.

rewrite ← *H5*.

auto.

assert (*getTLB v2* (*updateTLB v tlb MMU*) = *getTLB v2 MMU*).

rewrite ← *sameMMULookup*.

auto.

apply *H10*.

rewrite ← *H8*.

rewrite → *H12*.

apply *H9*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

```
auto.
Qed.
Theorem alwaysValidCFG : ∀ (c1 c2 : config),
(validCFG3 (getCFG c1) (getCMMU c1) (getTextStart c1) (getTextEnd c1)) ∧
(c1 ==> c2) →
(validCFG3 (getCFG c2) (getCMMU c2) (getTextStart c2) (getTextEnd c2)).
Proof.
intros.
destruct H as [H1 H2].
destruct H2.
destruct c.
auto.
auto.
auto.
auto.
auto.
auto.
simpl.
simpl in H1.
unfold validCFG3 in H1.
unfold validCFG3.
destruct H as [H2 H].
destruct H as [H3 H].
destruct H as [H4 H].
destruct H as [H5 H].
destruct H as [H6 H].
destruct H as [H7 H].
destruct H as [H8 H].
intros.
split.
```

`assert` $(cs \leq getPhysical\ (getTLB\ f\ MMU) \leq ce)$.

`apply` *H1*.

`auto`.

`rewrite` $\rightarrow$ *H4* `in` *H6*.

`rewrite` $\leftarrow$ *sameMMULookup*.

`auto`.

contradict H9.

`rewrite` $\rightarrow$ *H9*.

contradict H6.

`apply` *and_not_or*.

`split`.

`apply` *le_not_lt*.

`apply` *H6*.

`apply` *le_not_lt*.

`apply` *H6*.

`assert` $(cs \leq getPhysical\ (getTLB\ (pred\ f)\ MMU) \leq ce)$.

`apply` *H1*.

`auto`.

`rewrite` $\rightarrow$ *H4* `in` *H6*.

`rewrite` $\leftarrow$ *sameMMULookup*.

`auto`.

contradict H9.

`rewrite` $\rightarrow$ *H9*.

contradict H6.

`apply` *and_not_or*.

`split`.

`apply` *le_not_lt*.

`apply` *H6*.

`apply` *le_not_lt*.

`apply` *H6*.

`auto`.

```
auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

Qed.

Theorem alwaysInText : ∀ (c1 c2 : config) (v : nat),
(getTextStart c1 ≤ getPhysical (getTLB v (getCMMU c1)) ≤ getTextEnd c1) ∧
(c1 ==> c2) →
(getTextStart c2 ≤ getPhysical (getTLB v (getCMMU c2)) ≤ getTextEnd c2).
Proof.
intros.
destruct H as [H1 H].
destruct H.
destruct c.

auto.

auto.

auto.

auto.

auto.

auto.
```

simpl.

simpl in *H1*.

destruct *H* as [*H10 H*].

destruct *H* as [*H11 H*].

destruct *H* as [*H12 H*].

destruct *H* as [*H13 H*].

destruct *H* as [*H14 H*].

destruct *H* as [*H15 H*].

destruct *H* as [*H16 H*].

rewrite → *H12* in *H14*.

assert (*v* ≠ *v0*).

contradict H1.

rewrite → *H1*.

contradict H14.

apply *and_not_or*.

split.

apply *le_not_lt*.

apply *H14*.

apply *le_not_lt*.

apply *H14*.

rewrite ← *sameMMULookup*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

```
auto.

auto.

auto.

auto.

auto.

auto.

Qed.
```

B.10 ICText.v

```
Require Import Arith.

Require Import List.

Require Import Coq.Logic.Classical_Prop.

Require Export Semantics.

Definition ICInText (ic : InterruptContext) (cs ce : nat) (mmu : MMU) :=
  cs ≤ (getPhysical (getTLB (getICPC ic) mmu)) ≤ ce ∧
  cs ≤ (getPhysical (getTLB (pred (getICPC ic)) mmu)) ≤ ce.

Fixpoint ICListInText (icl : list InterruptContext) (cs ce : nat) (mmu : MMU) :=
  match icl with
  | nil ⇒ True
  | h :: t ⇒ (ICInText h cs ce mmu) ∧ (ICListInText t cs ce mmu)
  end.

Fixpoint ThreadICsInText (tl : ThreadList) (c : config) :=
  match tl with
  | nil ⇒ True
  | h :: t ⇒ (ICListInText (getThreadICL h) (getTextStart c) (getTextEnd c) (getCMMU c)) ∧ (ThreadIC-
sInText t c)
  end.

Fixpoint ThreadSICsInText (tl : ThreadList) (c : config) :=
  match tl with
  | nil ⇒ True
```

| $h :: t \Rightarrow (ICListInText\ (getThreadSICL\ h)\ (getTextStart\ c)\ (getTextEnd\ c)\ (getCMMU\ c)) \wedge (Thread-SICsInText\ t\ c)$

end.

Definition $AreAllThreadICsInText\ (tl : ThreadList)\ (c : config) :=$

$\forall\ (t : SVAThread), (In\ t\ tl) \rightarrow (ICListInText\ (getThreadICL\ t)\ (getTextStart\ c)\ (getTextEnd\ c)\ (getCMMU\ c)).$

Definition $AreAllThreadICsInText2\ (tl : ThreadList)\ (mmu : MMU)\ (cs\ ce : nat) :=$

$\forall\ (t : SVAThread), (In\ t\ tl) \rightarrow (ICListInText\ (getThreadICL\ t)\ cs\ ce\ mmu).$

Definition $AreAllThreadSICsInText\ (tl : ThreadList)\ (c : config) :=$

$\forall\ (t : SVAThread), (In\ t\ tl) \rightarrow (ICListInText\ (getThreadSICL\ t)\ (getTextStart\ c)\ (getTextEnd\ c)\ (getCMMU\ c)).$

Definition $addrNotInIC\ (v : nat)\ (ic : InterruptContext) :=$

$((getICPC\ ic) \neq v) \wedge ((pred\ (getICPC\ ic)) \neq v).$

Fixpoint $addrNotInICL\ (v : nat)\ (icl : ICStack) :=$

match icl with

| $nil \Rightarrow True$

| $h :: t \Rightarrow (addrNotInIC\ v\ h) \wedge (addrNotInICL\ v\ t)$

end.

Fixpoint $addrNotInTLICL\ (v : nat)\ (tl : ThreadList) :=$

match tl with

| $nil \Rightarrow True$

| $h :: t \Rightarrow (addrNotInICL\ v\ (getThreadICL\ h)) \wedge (addrNotInTLICL\ v\ t)$

end.

Definition $addrInICL\ (v : nat)\ (icl : ICStack) :=$

$\forall\ (ic : InterruptContext), (In\ ic\ icl) \rightarrow not\ (addrNotInIC\ v\ ic).$

Definition $noWritePCIC\ (ic : InterruptContext)\ (mmu : MMU) :=$

$(not\ (canWrite\ (getICPC\ ic)\ mmu)) \wedge$

$(not\ (canWrite\ (pred\ (getICPC\ ic))\ mmu)).$

Fixpoint $noWritePCICL\ (icl : list\ InterruptContext)\ (mmu : MMU) :=$

match icl with

```
  | nil ⇒ True
  | h :: t ⇒ (noWritePCIC h mmu) ∧ (noWritePCICL t mmu)
  end.

Fixpoint noWriteTLPC (tl : ThreadList) (mmu : MMU) :=
  match tl with
  | nil ⇒ True
  | h :: t ⇒ (noWritePCICL (getThreadICL h) mmu) ∧ (noWriteTLPC t mmu)
  end.

Theorem areAllImpliesAll: ∀ (tl : ThreadList) (c : config),
ThreadICsInText tl c → AreAllThreadICsInText tl c.
Proof.
intros.
induction tl.
unfold AreAllThreadICsInText.
intros.
unfold In in H0.
contradiction.
unfold ThreadICsInText in H.
destruct H as [H1 H2].
fold ThreadICsInText in H2.
unfold AreAllThreadICsInText.
intros.
destruct H.
rewrite ← H.
auto.
apply IHtl.
auto.
auto.
Qed.

Theorem areAllIffAll: ∀ (tl : ThreadList) (c : config),
```

ThreadICsInText tl c ↔ AreAllThreadICsInText tl c.

```
Proof.
intros.
split.
apply areAllImpliesAll.
intros.
induction tl.
simpl.
auto.
unfold ThreadICsInText.
split.
unfold AreAllThreadICsInText in H.
apply H.
simpl.
auto.
fold ThreadICsInText.
apply IHtl.
unfold AreAllThreadICsInText in H.
unfold AreAllThreadICsInText.
intros.
apply H.
unfold In.
right.
apply H0.
Qed.
```

```
Theorem SICareAllImpliesAll: ∀ (tl : ThreadList) (c : config),
```

ThreadSICsInText tl c → AreAllThreadSICsInText tl c.

```
Proof.
intros.
induction tl.
unfold AreAllThreadSICsInText.
```

```
intros.
unfold In in H0.
contradiction.
unfold ThreadSICsInText in H.
destruct H as [H1 H2].
fold ThreadSICsInText in H2.
unfold AreAllThreadSICsInText.
intros.
destruct H.
rewrite ← H.
auto.
apply IHtl.
auto.
auto.
Qed.
Theorem SICareAllIffAll: ∀ (tl : ThreadList) (c : config),
ThreadSICsInText tl c ↔ AreAllThreadSICsInText tl c.
Proof.
intros.
split.
apply SICareAllImpliesAll.
intros.
induction tl.
simpl.
auto.
unfold ThreadSICsInText.
split.
unfold AreAllThreadSICsInText in H.
apply H.
simpl.
```

```
auto.
```

fold ThreadSICsInText.

```
apply IHtl.
unfold AreAllThreadSICsInText in H.
unfold AreAllThreadSICsInText.
intros.
apply H.
unfold In.
```

right.

```
apply H0.
Qed.
```

```
Theorem ICListInTextImpliesICInText:
∀ (icl : list InterruptContext)
        (ic : InterruptContext)
        (cs ce : nat)
        (mmu : MMU),
(ICListInText icl cs ce mmu)
→
((In ic icl) → (ICInText ic cs ce mmu)).
Proof.
intros.
induction icl.
unfold In in H0.
```

contradiction.

```
unfold ICListInText in H.
destruct H as [H1 H2].
```

fold ICListInText `in` *H2.*

```
unfold In in H0.
destruct H0.
rewrite ← H.
```

```
auto.
apply IHicl.
auto.
auto.
Qed.
Theorem noWriteTLPCForAll:
∀ (mmu : MMU) (t : SVAThread) (tl : ThreadList),
(noWriteTLPC (t :: tl) mmu) → (noWriteTLPC tl mmu).
Proof.
intros c t.
induction tl.
intros.
simpl.
auto.
unfold noWriteTLPC.
split.
apply H.
fold noWriteTLPC.
fold noWriteTLPC in H.
apply IHtl.
simpl.
split.
apply H.
apply H.
Qed.
Theorem notInPCICL : ∀ (c : config) (v : nat),
(ThreadICsInText (getThreadList c) c) ∧
(getPhysical (getTLB v (getCMMU c)) < (getTextStart c) ∨
  (getTextEnd c) < getPhysical (getTLB v (getCMMU c)))
→
addrNotInTLICL v (getThreadList c).
```

```
Proof.
intros c.
induction (getThreadList).
intros.
unfold addrNotInTLICL.
auto.
intros.
destruct H as [H2 H1].
simpl in H1.
simpl in H2.
unfold addrNotInTLICL.
split.
induction getThreadICL.
unfold addrNotInICL.
auto.
unfold addrNotInICL.
split.
unfold addrNotInIC.
destruct H2 as [H2 H3].
unfold ICListInText in H2.
destruct H2 as [H2 H4].
fold ICListInText in H4.
unfold ICInText in H2.
destruct H2 as [H2 predH2].
destruct H2 as [H5 H6].
split.
contradict H1.
rewrite ← H1.
contradict H1.
destruct H1 as [H11 | H12].
apply lt_not_le in H11.
```

contradiction.

`apply` *lt_not_le* `in` *H12*.

contradiction.

`destruct` *predH2* `as` [*predH2 predH3*].

contradict H1.

`rewrite` ← *H1*.

contradict H1.

`destruct` *H1* `as` [*H11* | *H12*].

`apply` *lt_not_le* `in` *H11*.

contradiction.

`apply` *lt_not_le* `in` *H12*.

contradiction.

fold addrNotInTLICL.

`apply` *IHi*.

`split`.

`apply` *H2*.

`apply` *H2*.

fold addrNotInTLICL.

`apply` *IHt*.

`split`.

`apply` *H2*.

`auto`.

`Qed`.

`Theorem` *moreThreadICsInText*:

∀ (*mmu*: *MMU*)

(*ds* : *store*)

(*Reg* : *tm*)

(*PC SP* : *nat*)

(*cfg* : *list nat*)

(*cs ce* : *nat*)

(*st* : *list Stack*)

```
        (asid tid ntid : nat)
        (tl : ThreadList)
        (gvs gve : nat)
        (gl : list nat)
        (th : nat)
        (t : SVAThread),
ThreadICsInText tl
  (C mmu ds Reg PC SP cfg cs ce st asid tid ntid (t :: tl) gvs gve gl th) →
ThreadICsInText tl
  (C mmu ds Reg PC SP cfg cs ce st asid tid ntid tl gvs gve gl th).
Proof.
intros.
induction tl.
auto.
intros.
apply areAllIffAll.
apply areAllImpliesAll in H.
apply H.
Qed.
Theorem addThreadICsInText: ∀ (c : config) (t : SVAThread),
(ThreadICsInText (t :: (getThreadList c)) (setThreadList (t :: getThreadList c) c))
→
(ThreadICsInText (getThreadList c) c).
Proof.
intros.
unfold ThreadICsInText in H.
destruct H as [H1 H2].
fold ThreadICsInText in H2.
destruct c.
simpl in H1.
simpl in H2.
```

unfold *ICListInText* in *H1*.

unfold *ThreadICsInText*.

simpl.

fold ThreadICsInText.

apply *moreThreadICsInText* with $(t := t)$.

auto.

Qed.

Theorem *threadOnICListInText*:

$\forall\ (tl : ThreadList)\ (cs\ ce : nat)\ (mmu : MMU)\ (id : nat)\ (pc : nat),$

$ICListInText\ (getThreadICL\ (getThread\ id\ tl))\ cs\ ce\ mmu \rightarrow$

$ICListInText\ (getThreadICL\ (threadOnCPU\ pc\ (getThread\ id\ tl)))\ cs\ ce\ mmu.$

Proof.

intros.

generalize dependent *id*.

induction *tl*.

intros.

destruct *id*.

auto.

auto.

intros.

destruct *id*.

unfold *getThread*.

simpl.

unfold *getThread* in *H*.

simpl in *H*.

destruct *a*.

simpl.

simpl in *H*.

auto.

unfold *getThread* in *H*.

simpl in *H*.

unfold *getThread*.

simpl.

apply *IHtl*.

auto.

Qed.

Theorem *threadOffICListInText*:

∀ (*tl* : *ThreadList*) (*cs ce* : *nat*) (*mmu* : *MMU*) (*id* : *nat*),

ICListInText (*getThreadICL* (*getThread id tl*)) *cs ce mmu* →

ICListInText (*getThreadICL* (*threadOffCPU* (*getThread id tl*))) *cs ce mmu*.

Proof.

intros.

generalize dependent *id*.

induction *tl*.

intros.

destruct *id*.

auto.

auto.

intros.

destruct *id*.

unfold *getThread*.

simpl.

unfold *getThread* in *H*.

simpl in *H*.

destruct *a*.

simpl.

simpl in *H*.

auto.

unfold *getThread* in *H*.

simpl in *H*.

unfold *getThread*.

simpl.

apply *IHtl*.

auto.

Qed.

Theorem *threadOnSICListInText*:

$\forall$ (*tl* : *ThreadList*) (*cs ce* : *nat*) (*mmu* : *MMU*) (*id* : *nat*) (*pc* : *nat*),

ICListInText (*getThreadSICL* (*getThread id tl*)) *cs ce mmu* $\rightarrow$

ICListInText (*getThreadSICL* (*threadOnCPU pc* (*getThread id tl*))) *cs ce mmu*.

Proof.

intros.

generalize dependent *id*.

induction *tl*.

intros.

destruct *id*.

auto.

auto.

intros.

destruct *id*.

unfold *getThread*.

simpl.

unfold *getThread* in *H*.

simpl in *H*.

destruct *a*.

simpl.

simpl in *H*.

auto.

unfold *getThread* in *H*.

simpl in *H*.

unfold *getThread*.

simpl.

apply *IHtl*.

auto.

```
Qed.

Theorem threadOffSICListInText:
∀ (tl : ThreadList) (cs ce : nat) (mmu : MMU) (id : nat),
ICListInText (getThreadSICL (getThread id tl)) cs ce mmu →
ICListInText (getThreadSICL (threadOffCPU (getThread id tl))) cs ce mmu.
Proof.
intros.
generalize dependent id.
induction tl.
intros.
destruct id.
auto.
auto.
intros.
destruct id.
unfold getThread.
simpl.
unfold getThread in H.
simpl in H.
destruct a.
simpl.
simpl in H.
auto.
unfold getThread in H.
simpl in H.
unfold getThread.
simpl.
apply IHtl.
auto.
Qed.
```

B.11 ICProofs.v

```
Require Import Arith.

Require Import List.

Require Import Coq.Logic.Classical_Prop.

Require Export Semantics.

Require Export ICText.

Definition goodPCIC (ic : InterruptContext)
                    (cfg : list nat)
                    (mmu : MMU)
                    (ds : store) :=
  (In (getICPC ic) cfg) ∨
  ((vLookup ((getICPC ic) - 1) mmu ds) = trap) ∨
  ((getICPC ic) = 0).

Fixpoint goodPCICL (icl : ICStack)
                   (cfg : list nat)
                   (mmu : MMU)
                   (ds : store) :=
   match icl with
   | nil ⇒ True
   | h :: t ⇒ (goodPCIC h cfg mmu ds) ∧ (goodPCICL t cfg mmu ds)
 end.

Definition goodPCInIC (t : SVAThread)
                      (cfg : list nat)
                      (mmu : MMU)
                      (ds : store) :=
  (goodPCICL (getThreadICL t) cfg mmu ds).

Definition goodPCInSIC (t : SVAThread)
                       (cfg : list nat)
                       (mmu : MMU)
                       (ds : store) :=
```

```
  (goodPCICL (getThreadSICL t) cfg mmu ds).

Fixpoint goodPCInTL (tl : ThreadList) (cfg : list nat) (mmu : MMU) (ds : store) :=
  match tl with
  | nil ⇒ True
  | h :: t ⇒ (goodPCInIC h cfg mmu ds) ∧ (goodPCInTL t cfg mmu ds)
  end.

Fixpoint goodSPCInTL (tl : ThreadList) (cfg : list nat) (mmu : MMU) (ds : store) :=
  match tl with
  | nil ⇒ True
  | h :: t ⇒ (goodPCInSIC h cfg mmu ds) ∧ (goodSPCInTL t cfg mmu ds)
  end.

Definition goodPCInConfigIC (c : config) :=
  goodPCInTL (getThreadList c) (getCFG c) (getCMMU c) (getStore c).

Definition goodPCInConfigSIC (c : config) :=
  goodSPCInTL (getThreadList c) (getCFG c) (getCMMU c) (getStore c).

Theorem popInText: ∀ (icl : list InterruptContext)
                     (cs ce : nat)
                     (mmu : MMU),
ICListInText icl cs ce mmu →
ICListInText (pop icl) cs ce mmu.
Proof.
intros.
induction icl.
auto.
unfold ICListInText in H.
destruct H as [H1 H2].
fold ICListInText in H2.
unfold pop.
auto.
Qed.
```

```
Theorem addPCInICThread: ∀ (t : SVAThread) (tid f : nat) (tl : ThreadList) (cfg : list nat) (mmu :
MMU) (ds : store),
(goodPCInIC t cfg mmu ds) ∧ goodPCInTL tl cfg mmu ds →
goodPCInTL (t :: tl) cfg mmu ds.
Proof.
intros.
destruct H as [H1 H2].
induction tl.
unfold goodPCInTL.
split.
auto.
auto.
unfold goodPCICL.
split.
auto.
split.
destruct H2 as [H2 H3].
auto.
fold goodPCInTL.
destruct H2 as [H2 H3].
auto.
Qed.
Theorem replacePCInICThread: ∀ (t : SVAThread) (tl : ThreadList) (cfg : list nat) (mmu : MMU) (ds :
store) (tid : nat),
(goodPCInIC t cfg mmu ds) ∧ goodPCInTL tl cfg mmu ds →
goodPCInTL (updateThread tid t tl) cfg mmu ds.
Proof.
intros t tl cfg mmu ds.
induction tl.
intros.
destruct H as [H1 H2].
```

```
induction tid.
auto.
auto.
intros.
destruct H as [H2 H3].
destruct tid.
simpl.
unfold goodPCInTL in H3.
destruct H3 as [H3 H4].
auto.
simpl.
destruct H3 as [H3 H4].
auto.
Qed.

Theorem goodPCInItop : ∀ (icl : ICStack) (cfg : list nat) (mmu : MMU) (ds : store),
  goodPCICL icl cfg mmu ds → goodPCIC (itop icl) cfg mmu ds.
Proof.
intros.
destruct icl.
unfold itop.
simpl.
unfold goodPCIC.
right.
right.
auto.
unfold itop.
simpl.
destruct H as [H1 H2].
auto.
Qed.
```

```
Theorem pcInICLWrite:
∀ (icl : ICStack) (cfg : list nat) (mmu : MMU) (ds : store) (n : tm) (v : nat) (cs ce : nat),
((addrNotInICL v icl) ∧ (goodPCICL icl cfg mmu ds) ∧
(ICListInText icl cs ce mmu) ∧
(not (cs ≤ (getPhysical (getTLB v mmu)) ≤ ce))) →
(goodPCICL icl cfg mmu (replace (getPhysical (getTLB v mmu)) n ds)).
Proof.
intros.
generalize dependent n.
generalize dependent v.
induction icl.
intros.
unfold goodPCICL.
auto.
intros.
destruct H as [H1 H2].
unfold goodPCICL.
split.
unfold addrNotInICL in H1.
destruct H1 as [H1 H3].
fold addrNotInICL in H3.
unfold addrNotInIC in H1.
unfold goodPCIC.
destruct H1 as [H4 H5].
unfold goodPCICL in H2.
destruct H2 as [H1 H2].
fold goodPCICL in H2.
unfold goodPCIC in H1.
destruct H1 as [H1 H6].
destruct H1 as [H10 | H11].
left.
```

```
auto.
destruct H11 as [H11 | H12].
right.
left.
unfold vLookup.
unfold vLookup in H11.
rewrite ← sameRead.
rewrite → H11.
auto.
rewrite ← pred_of_minus.
destruct H2 as [H2 H7].
unfold ICListInText in H2.
destruct H2 as [H2 H12].
fold ICListInText in H12.
unfold ICInText in H2.
destruct H2 as [H20 H21].
contradict H21.
rewrite → H21.
apply H7.
right.
right.
auto.
fold goodPCICL.
apply IHicl.
unfold addrNotInICL in H1.
split.
apply H1.
unfold goodPCICL in H2.
split.
apply H2.
split.
```

```
apply H2.
apply H2.
Qed.

Theorem pcInTLWrite :
∀ (tl : ThreadList) (cfg : list nat) (mmu : MMU) (ds : store) (n : tm) (v : nat) (cs ce : nat),
((addrNotInTLICL v tl) ∧ (goodPCInTL tl cfg mmu ds) ∧
(AreAllThreadICsInText2 tl mmu cs ce) ∧
(not (cs ≤ (getPhysical (getTLB v mmu)) ≤ ce)))
→
goodPCInTL tl cfg mmu (replace (getPhysical (getTLB v mmu)) n ds).
Proof.
intros tl cfg mmu ds.
induction tl.

intros.
unfold goodPCInTL.
auto.

intros.
destruct H as [H1 H2].
unfold goodPCInTL.
split.
apply pcInICLWrite with (cs := cs) (ce := ce).
split.
unfold addrNotInTLICL in H1.
apply H1.
unfold goodPCInTL in H2.
split.
apply H2.
split.
apply H2.
unfold In.
left.
```

```
auto.
apply H2.
fold goodPCInTL.
destruct H2 as [H2 H3].
assert (AreAllThreadICsInText2 tl mmu cs ce).
unfold AreAllThreadICsInText2.
unfold AreAllThreadICsInText2 in H3.
intros.
apply H3.
unfold In.
right.
auto.
apply IHtl with (cs := cs) (ce := ce).
split.
unfold addrNotInTLICL in H1.
apply H1.
unfold goodPCInTL in H2.
destruct H2 as [H2 H4].
split.
apply H4.
split.
apply H.
apply H3.
Qed.

Theorem swapOnPCIC: ∀ (tl : ThreadList) (cfg : list nat) (mmu : MMU) (ds : store) (pc : nat) (tid : nat),
goodPCInTL tl cfg mmu ds →
goodPCInIC (threadOnCPU pc (getThread tid tl)) cfg mmu ds.
Proof.
intros tl cfg mmu ds pc.
induction tl.
destruct tid.
```

```
auto.
auto.
intros.
destruct H as [H1 H2].
fold goodPCInTL in H2.
destruct tid.
unfold getThread.
simpl.
destruct a.
unfold threadOnCPU.
auto.
unfold getThread.
simpl.
unfold getThread in IHtl.
auto.
Qed.

Theorem swapOffPCIC: ∀ (tl : ThreadList) (cfg : list nat) (mmu : MMU) (ds : store) (tid : nat),
goodPCInTL tl cfg mmu ds →
goodPCInIC (threadOffCPU (getThread tid tl)) cfg mmu ds.
Proof.
intros tl cfg mmu ds.
induction tl.
destruct tid.
auto.
auto.
intros.
destruct H as [H1 H2].
fold goodPCInTL in H2.
destruct tid.
unfold getThread.
```

```
simpl.
destruct a.
unfold threadOffCPU.
auto.
unfold getThread.
simpl.
unfold getThread in IHtl.
auto.
Qed.
Theorem popGoodPCICL: ∀ (tl : ThreadList) (cfg : list nat) (mmu : MMU) (ds : store) (tid : nat),
goodPCICL (getThreadICList tid tl) cfg mmu ds →
goodPCICL (pop (getThreadICList tid tl)) cfg mmu ds.
Proof.
intros tl cfg mmu ds.
induction tl.
intros.
destruct tid.
auto.
auto.
intros.
destruct tid.
unfold getThreadICList.
unfold getThread.
simpl.
unfold getThreadICList in H.
unfold getThread in H.
simpl in H.
destruct getThreadICL.
auto.
unfold pop.
unfold goodPCICL in H.
```

destruct *H* as [*H1 H2*].

auto.

unfold *getThreadICList*.

unfold *getThread*.

simpl.

unfold *getThreadICList* in *H*.

unfold *getThread* in *H*.

simpl in *H*.

apply *IHtl*.

auto.

Qed.

Theorem *GoodPCICL*: ∀ (*tl* : *ThreadList*)

(*cfg* : *list nat*)

(*mmu* : *MMU*)

(*ds* : *store*)

(*tid* : *nat*),

goodPCInTL tl cfg mmu ds → *goodPCICL* (*getThreadICList tid tl*) *cfg mmu ds*.

Proof.

intros *tl cfg mmu ds*.

induction *tl*.

intros.

destruct *tid*.

auto.

auto.

intros.

destruct *tid*.

unfold *getThreadICList*.

unfold *getThread*.

simpl.

destruct *H* as [*H1 H2*].

auto.

unfold *getThreadICList*.

unfold *getThread*.

simpl.

destruct *H* as [*H1 H2*].

fold goodPCInTL in *H2*.

apply *IHtl*.

auto.

Qed.

Theorem *GoodPCSICL*: ∀ (*tl* : *ThreadList*)

(*cfg* : *list nat*)

(*mmu* : *MMU*)

(*ds* : *store*)

(*tid* : *nat*),

goodSPCInTL tl cfg mmu ds → *goodPCICL* (*getThreadSICList tid tl*) *cfg mmu ds*.

Proof.

intros *tl cfg mmu ds*.

induction *tl*.

intros.

destruct *tid*.

auto.

auto.

intros.

destruct *tid*.

unfold *getThreadSICList*.

unfold *getThread*.

simpl.

destruct *H* as [*H1 H2*].

auto.

unfold *getThreadSICList*.

unfold *getThread*.

simpl.

```
destruct H as [H1 H2].
fold goodSPCInTL in H2.
apply IHtl.
auto.
Qed.

Theorem updateMMUICL : ∀ (t : SVAThread) (tlb : TLBTy) (cfg : list nat)
(mmu : MMU) (ds : store) (v : nat),
(addrNotInICL v (getThreadICL t)) ∧
(goodPCInIC t cfg mmu ds) →
(goodPCInIC t cfg (updateTLB v tlb mmu) ds).
Proof.
intros t tlb cfg mmu ds.
unfold goodPCInIC.
induction getThreadICL.

intros.
unfold goodPCICL.
auto.

intros.
destruct H as [H1 H2].
unfold goodPCICL in H2.
destruct H2 as [H2 H3].
fold goodPCICL in H3.
unfold goodPCICL.
split.

unfold goodPCIC.
unfold goodPCIC in H2.
destruct H2 as [H4 | H5 ].
left.
auto.
destruct H5 as [H5 | H6].
right.
```

left.

`unfold` *vLookup*.

`unfold` *vLookup* `in` *H5*.

`rewrite` ← *sameMMULookup*.

`auto`.

`unfold` *addrNotInICL* `in` *H1*.

`destruct` *H1* `as` [*H1 H2*].

`unfold` *addrNotInIC* `in` *H1*.

`destruct` *H1* `as` [*H1 H4*].

`rewrite` ← *pred_of_minus*.

`auto`.

`auto`.

fold goodPCICL.

`apply` *IHi*.

`split`.

`destruct` *H1* `as` [*H1 H4*].

`auto`.

`auto`.

`Qed`.

`Theorem` *updateMMUCIC* : ∀ (*tl* : *ThreadList*) (*tlb* : *TLBTy*) (*cfg* : *list nat*) (*mmu* : *MMU*) (*ds* : *store*) (*v* :*nat*),

(*addrNotInTLICL v tl*) ∧ (*goodPCInTL tl cfg mmu ds*) →

(*goodPCInTL tl cfg* (*updateTLB v tlb mmu*) *ds*).

`Proof`.

`intros` *tl tbl cfg mmu ds*.

`induction` *tl*.

`intros`.

`unfold` *goodPCInTL*.

`auto`.

`intros`.

`destruct` *H* `as` [*H1 H2*].

unfold *goodPCInTL* in *H2*.

destruct *H2* as [*H2 H3*].

fold goodPCInTL in *H3*.

unfold *goodPCInTL*.

split.

apply *updateMMUICL*.

split.

unfold *addrNotInTLICL* in *H1*.

apply *H1*.

auto.

fold goodPCInTL.

apply *IHtl*.

split.

unfold *addrNotInTLICL* in *H1*.

destruct *H1* as [*H1 H4*].

fold addrNotInTLICL in *H4*.

auto.

auto.

Qed.

Theorem *pcInIC*: $\forall$ (*c1 c2* : *config*),

(*c1* ==> *c2*) $\wedge$

(*goodPCInConfigIC c1*) $\wedge$

(*validConfig c1*) $\wedge$

(*textNotWriteable c1*) $\wedge$

(*ThreadICsInText* (*getThreadList c1*) *c1*) $\wedge$

((*getTextStart c1*) $\leq$ (*getPhysical* (*getTLB* (*getPC c1*) (*getCMMU c1*))) $\leq$ (*getTextEnd c1*)) $\wedge$

(*validCFG c1* (*getCFG c1*)) $\wedge$

(*goodPCInConfigSIC c1*) $\wedge$

(*noWriteTLPC* (*getThreadList c1*) (*getCMMU c1*))

$\rightarrow$ (*goodPCInConfigIC c2*).

Proof.

intros.

destruct *H* as [*step inv*].

destruct *inv* as [*inv valid*].

destruct *valid* as [*valid norw*].

unfold *goodPCInConfigIC*.

unfold *goodPCInConfigIC* in *inv*.

destruct *step*.

destruct *c*.

simpl in *inv*.

auto.

auto.

simpl.

simpl in *inv*.

simpl in *norw*.

apply *pcInTLWrite* with (*cs* := *cs*) (*ce* := *ce*).

split.

destruct *norw* as [*norw1 norw2*].

destruct *norw2* as [*norw3 norw2*].

destruct *norw2* as [*norw4 norw2*].

destruct *norw2* as [*norw5 norw2*].

destruct *norw2* as [*norw6 norw2*].

induction *tl*.

unfold *addrNotInTLICL*.

auto.

unfold *addrNotInTLICL*.

split.

destruct *inv* as [*inv1 inv2*].

fold goodPCInTL in *inv2*.

unfold *goodPCInIC* in *inv1*.

unfold *noWriteTLPC* in *norw2*.

destruct *norw2* as [*norw7 norw2*].

induction *getThreadICL*.

unfold *addrNotInICL*.

auto.

unfold *addrNotInICL*.

split.

unfold *addrNotInIC*.

unfold *noWritePCICL* in *norw7*.

destruct *norw7* as [*norw7 norw8*].

fold noWriteTLPC in *norw2*.

fold noWritePCICL in *norw8*.

unfold *noWritePCIC* in *norw7*.

simpl in *norw7*.

destruct *norw7* as [*wr1 wr2*].

split.

contradict wr1.

rewrite → *wr1*.

apply *H*.

contradict wr2.

rewrite → *wr2*.

apply *H*.

fold addrNotInICL.

apply *IHi*.

unfold *goodPCICL* in *inv1*.

apply *inv1*.

fold noWriteTLPC in *IHi*.

unfold *goodPCICL* in *inv1*.

destruct *inv1* as [*inv1 inv3*].

fold goodPCICL in *inv3*.

unfold *noWritePCICL* in *norw7*.

apply *norw7*.

fold noWriteTLPC.

auto.

fold addrNotInTLICL.

apply *IHtl.*

unfold *goodPCInTL* in *inv.*

apply *inv.*

auto.

apply *addThreadICsInText* with ($t := a$) ($c := (C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th)$).

simpl.

auto.

unfold *validCFG.*

simpl.

unfold *validCFG* in *norw5.*

apply *norw5.*

unfold *goodPCInConfigSIC* in *norw6.*

simpl in *norw6.*

unfold *goodPCInConfigSIC.*

simpl.

apply *norw6.*

destruct *inv* as [*inv1 inv2*].

fold goodPCInTL in *inv2.*

unfold *validConfig* in *valid.*

unfold *ThreadICsInText* in *norw3.*

destruct *norw3* as [*norw3a norw3b*].

fold ThreadICsInText in *norw3b.*

unfold *validCFG* in *norw5.*

simpl in *norw5.*

unfold *goodPCInConfigSIC* in *norw6.*

simpl in *norw6.*

unfold *noWriteTLPC* in *norw2.*

destruct *norw2* as [*norw2a norw2b*].

fold noWriteTLPC `in` *norw2b*.

`auto`.

`split`.

`auto`.

`rewrite` → *areAllIffAll* `in` *norw*.

`unfold` *AreAllThreadICsInText* `in` *norw*.

`simpl in` *norw*.

`split`.

`unfold` *AreAllThreadICsInText2*.

`apply` *norw*.

`destruct` *norw* `as` [*norw1 norw2*].

`destruct` *H* `as` [*H1 H*].

`destruct` *H* `as` [*H2 H*].

`destruct` *H* `as` [*H3 H*].

contradict H3.

`apply` *norw1*.

`auto`.

`auto`.

`auto`.

`auto`.

`simpl`.

`simpl in` *inv*.

`simpl in` *norw*.

`destruct` *norw* `as` [*H1 H2*].

`destruct` *H2* `as` [*H2 H3*].

`destruct` *H3* `as` [*H3 H4*].

`destruct` *H4* `as` [*H4 H5*].

`apply` *updateMMUCIC*.

`split`.

`destruct` *H* `as` [*H10 H11*].

`destruct` *H11* `as` [*H12 H11*].

destruct *H11* as [*H13 H11*].

destruct *H11* as [*H14 H11*].

destruct *H11* as [*H15 H11*].

destruct *H11* as [*H16 H11*].

rewrite → *H13* in *H15*.

assert (*tl* = *getThreadList* ((*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*))).

simpl.

auto.

apply *notInPCICL* with (*c* := (*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*)).

simpl.

split.

auto.

apply *H15*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

simpl.

simpl in *inv*.

simpl in *norw*.

destruct *norw* as [*H1 norw*].

destruct *norw* as [*H2 norw*].

destruct *norw* as [*H3 norw*].

destruct *norw* as [*H4 norw*].

destruct *norw* as [*H5 norw*].

assert (*AreAllThreadICsInText tl* (*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*)).

apply *areAllImpliesAll*.

auto.

apply *replacePCInICThread*.

split.

unfold *goodPCInIC*.

unfold *getThreadICL*.

unfold *goodPCICL*.

split.

apply *goodPCInItop*.

unfold *goodPCInTL* in *inv*.

apply *GoodPCICL*.

auto.

auto.

auto.

simpl.

simpl in *inv*.

apply *replacePCInICThread*.

split.

apply *swapOnPCIC*.

auto.

apply *replacePCInICThread*.

split.

apply *swapOffPCIC*.

auto.

auto.

simpl.

apply *replacePCInICThread*.

split.

unfold *push*.

unfold *goodPCInIC*.

simpl.

split.

unfold *goodPCIC*.

right.

simpl.

rewrite ← *minus_n_O*.

destruct *H* as [*H1* *H2*].

unfold *vLookup*.

unfold *vaLookup* in *H2*.

left.

apply *H2*.

unfold *getThreadList* in *inv*.

unfold *getCMMU* in *inv*.

unfold *getCFG* in *inv*.

simpl in *inv*.

unfold *goodPCInTL* in *inv*.

apply *GoodPCICL*.

auto.

auto.

simpl.

apply *replacePCInICThread*.

split.

unfold *goodPCInIC*.

simpl.

apply *popGoodPCICL*.

apply *GoodPCICL*.

auto.

auto.

simpl.

simpl in *inv*.

induction *tl*.

destruct *tid*.

unfold *pushSIC*.

auto.

unfold *pushSIC*.

auto.

simpl in *IHtl*.

destruct *tid*.

simpl.

auto.

destruct *inv* as [*inv1 inv2*].

fold goodPCInTL in *inv2*.

simpl.

split.

auto.

auto.

simpl.

apply *replacePCInICThread*.

split.

simpl.

unfold *goodPCInIC*.

unfold *getThreadICL*.

apply *GoodPCICL*.

auto.

auto.

simpl.

unfold *popSIC*.

apply *replacePCInICThread*.

split.

unfold *goodPCInIC*.

simpl.

split.

apply *goodPCInItop*.

apply *GoodPCSICL*.

simpl in *norw*.

destruct *norw* as [*norw H1*].

destruct *H1* as [*H1 H2*].

destruct *H2* as [*H2 H3*].

destruct *H3* as [*H3 H4*].

destruct *H4* as [*H4 H5*].

unfold *goodPCInConfigSIC* in *H4*.

simpl in *H4*.

auto.

apply *popGoodPCICL*.

apply *GoodPCICL*.

simpl in *inv*.

auto.

auto.

simpl.

simpl in *inv*.

simpl *norw*.

unfold *pushIC*.

apply *replacePCInICThread*.

split.

unfold *push*.

unfold *goodPCInIC*.

simpl.

split.

unfold *goodPCIC*.

unfold *getICPC*.

destruct *H* as [*H1 H2*].

destruct *H2* as [*H2 H3*].

left.

auto.

apply *popGoodPCICL*.

```
apply GoodPCICL.
auto.
auto.
Qed.
Theorem updateThreadICL:
∀ (t : SVAThread) (tl : ThreadList) (tid : nat) (c : config),
(AreAllThreadICsInText tl c) ∧
(tid < length tl) ∧
(ICListInText (getThreadICL t) (getTextStart c) (getTextEnd c) (getCMMU c))
→
(AreAllThreadICsInText (updateThread tid t tl) c).
Proof.
intros.
destruct H as [H1 H2].
destruct H2 as [H2 H3].
generalize dependent tid.
induction tl.
intros.
destruct tid.
auto.
auto.
intros.
destruct tid.
unfold updateThread.
apply areAllImpliesAll.
unfold ThreadICsInText.
split.
auto.
fold ThreadICsInText.
apply areAllIffAll in H1.
unfold ThreadICsInText in H1.
```

destruct *H1* as [*H1 H4*].

fold ThreadICsInText in *H4*.

auto.

simpl in *H2*.

apply *lt_S_n* in *H2*.

unfold *updateThread*.

apply *areAllImpliesAll*.

unfold *ThreadICsInText*.

split.

apply *H1*.

unfold *In*.

auto.

fold ThreadICsInText.

fold updateThread.

apply *areAllIffAll*.

apply *IHtl*.

apply *areAllIffAll* in *H1*.

unfold *ThreadICsInText* in *H1*.

destruct *H1* as [*H1 H4*].

fold ThreadICsInText in *H4*.

apply *areAllIffAll*.

auto.

auto.

Qed.

Theorem *updateThreadSICL*:

$\forall$ (t : *SVAThread*) (tl : *ThreadList*) (tid : *nat*) (c : *config*),

(*AreAllThreadSICsInText tl c*) $\wedge$

($tid <$ *length tl*) $\wedge$

(*ICListInText* (*getThreadSICL t*) (*getTextStart c*) (*getTextEnd c*) (*getCMMU c*))

$\rightarrow$

(*AreAllThreadSICsInText* (*updateThread tid t tl*) *c*).

```
Proof.
intros.
destruct H as [H1 H2].
destruct H2 as [H2 H3].
generalize dependent tid.
induction tl.
intros.
destruct tid.
auto.
auto.
intros.
destruct tid.
unfold updateThread.
apply SICareAllImpliesAll.
unfold ThreadICsInText.
split.
auto.
fold ThreadICsInText.
apply SICareAllIffAll in H1.
unfold ThreadSICsInText in H1.
destruct H1 as [H1 H4].
fold ThreadSICsInText in H4.
auto.
simpl in H2.
apply lt_S_n in H2.
unfold updateThread.
apply SICareAllImpliesAll.
unfold ThreadSICsInText.
split.
apply H1.
unfold In.
```

```
auto.
```

fold ThreadSICsInText.

fold updateThread.

```
apply SICareAllIffAll.
apply IHtl.
apply SICareAllIffAll in H1.
unfold ThreadSICsInText in H1.
destruct H1 as [H1 H4].
```

fold ThreadSICsInText `in` *H4.*

```
apply SICareAllIffAll.
auto.
auto.
Qed.
```

`Theorem` *updateMMUICInText*:

$\forall$ (cs ce : nat) (mmu : MMU) (tlb : $TLBTy$) (v : nat) (ic : $InterruptContext$),

($addrNotInIC$ v ic) $\wedge$ ($ICInText$ ic cs ce mmu) $\rightarrow$ ($ICInText$ ic cs ce ($updateTLB$ v tlb mmu)).

```
Proof.
intros.
destruct H as [H1 H2].
unfold ICInText.
split.
unfold addrNotInIC in H1.
destruct H1 as [H3 H4].
rewrite ← sameMMULookup.
apply H2.
auto.
rewrite ← sameMMULookup.
apply H2.
apply H1.
Qed.
```

`Theorem` *updateMMUICListInText*:

$\forall$ (cs ce : nat) (mmu : MMU) (tlb : $TLBTy$) (v : nat) (icl : $list$ $InterruptContext$),

($addrNotInICL$ v icl) $\wedge$ ($ICListInText$ icl cs ce mmu) $\rightarrow$ ($ICListInText$ icl cs ce ($updateTLB$ v tlb mmu)).

`Proof`.

`intros`.

`destruct` H `as` [$H1$ $H2$].

`induction` icl.

`auto`.

`unfold` $addrNotInICL$ `in` $H1$.

`destruct` $H1$ `as` [$H1$ $H3$].

$fold$ $addrNotInICL$ `in` $H3$.

`unfold` $ICListInText$ `in` $H2$.

`destruct` $H2$ `as` [$H2$ $H4$].

$fold$ $ICListInText$ `in` $H4$.

`unfold` $ICListInText$.

`split`.

`apply` $updateMMUICInText$.

`auto`.

$fold$ $ICListInText$.

`apply` $IHicl$.

`auto`.

`auto`.

`Qed`.

`Theorem` $goodSICImpliesOut$: $\forall$ (v cs ce : nat) (mmu : MMU) (icl : $list$ $InterruptContext$),

($ICListInText$ icl cs ce mmu) $\wedge$

($getPhysical$ ($getTLB$ v mmu) $<$ cs $\vee$ ce $<$ $getPhysical$ ($getTLB$ v mmu))

$\rightarrow$

($addrNotInICL$ v icl).

`Proof`.

`intros`.

`destruct` H `as` [$H1$ $H2$].

`induction` icl.

auto.

unfold *ICListInText* in *H1*.

destruct *H1* as [*H3 H4*].

fold ICListInText in *H4*.

unfold *addrNotInICL*.

fold addrNotInICL.

split.

destruct *a*.

unfold *addrNotInIC*.

simpl.

unfold *ICInText* in *H3*.

simpl in *H3*.

destruct *H3* as [*H5 H6*].

split.

contradict H2.

rewrite ← *H2*.

apply *and_not_or*.

split.

apply *le_not_lt*.

apply *H5*.

apply *le_not_lt*.

apply *H5*.

contradict H2.

rewrite ← *H2*.

apply *and_not_or*.

split.

apply *le_not_lt*.

apply *H6*.

apply *le_not_lt*.

apply *H6*.

apply *IHicl*.

auto.

Qed.

Theorem *goodTLImpliesGoodICL*:

$\forall$ (*t* : *SVAThread*) (*tl* : *ThreadList*) (*v* : *nat*),

(*In t tl*) $\wedge$ (*addrNotInTLICL v tl*) $\rightarrow$ *addrNotInICL v* (*getThreadICL t*).

Proof.

intros.

destruct *H* as [*H1 H2*].

induction *tl*.

simpl in *H1*.

contradiction.

unfold *addrNotInTLICL* in *H2*.

destruct *H2* as [*H2 H3*].

fold addrNotInTLICL in *H3*.

simpl in *H1*.

destruct *H1*.

rewrite $\leftarrow$ *H*.

auto.

apply *IHtl*.

auto.

auto.

Qed.

Theorem *stayICInText*: $\forall$ (*c1 c2* : *config*),

(*c1* ==> *c2*) $\wedge$

(*pcInText c1*) $\wedge$

(*AreAllThreadICsInText* (*getThreadList c1*) (*c1*)) $\wedge$

(*validThreadIDs c1*) $\wedge$

(*validCFG c1* (*getCFG c1*)) $\wedge$

(*In* (*getThread* (*getCurrThread c1*) (*getThreadList c1*)) (*getThreadList c1*)) ∧

(*textMappedLinear c1*) ∧

(*AreAllThreadSICsInText* (*getThreadList c1*) (*c1*))

→

(*AreAllThreadICsInText* (*getThreadList c2*) (*c2*)).

`Proof.`

`intros.`

`destruct` *H* `as` [*H1 H2*].

`destruct` *H2* `as` [*H2 H3*].

`destruct` *H3* `as` [*H3 H4*].

`destruct` *H4* `as` [*H4 cfg*].

`destruct` *cfg* `as` [*cfg inThread*].

`destruct` *inThread* `as` [*inThread tml*].

`destruct` *tml* `as` [*tml sic*].

`destruct` *H1.*

`destruct` *c.*

`auto.`

`auto.`

`auto.`

`auto.`

`auto.`

`auto.`

`simpl.`

`unfold` *AreAllThreadICsInText.*

`simpl.`

`simpl in` *H3.*

`simpl in` *H2.*

`simpl in` *inThread.*

`simpl in` *tml.*

`simpl in` *sic.*

`simpl in` *cfg.*

unfold *AreAllThreadICsInText* in *H3*.

simpl in *H3*.

intros.

apply *updateMMUICListInText*.

split.

assert (*addrNotInTLICL v* (*getThreadList* (*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*))).

apply *notInPCICL*.

simpl.

split.

apply *areAllIffAll*.

auto.

destruct *H* as [*I1 H*].

destruct *H* as [*I2 H*].

destruct *H* as [*I3 H*].

destruct *H* as [*I4 H*].

destruct *H* as [*I5 H*].

rewrite → *I3* in *I5*.

auto.

unfold *getThreadList* in *H1*.

apply *goodTLImpliesGoodICL* with (*tl* := *tl*).

split.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

```
simpl.
simpl in H3.
apply updateThreadICL.
split.
simpl.
unfold AreAllThreadICsInText.
simpl.
unfold AreAllThreadICsInText in H3.
simpl in H3.
apply H3.
split.
destruct H as [H H1].
destruct H1 as [H1 H5].
destruct H4 as [H4 H6].
auto.
simpl.
unfold getThreadICList.
assert (In (getThread tid tl) tl).
apply nth_In.
apply H4.
split.
unfold ICInText.
apply ICListInTextImpliesICInText with (icl := (getThreadICL (getThread tid tl))).
apply H3.
apply H0.
auto.
apply nth_In.
apply H.
auto.
simpl in H2.
simpl in H3.
```

simpl in *inThread*.

simpl in *cfg*.

apply *updateThreadICL*.

split.

simpl.

apply *updateThreadICL*.

split.

simpl.

unfold *AreAllThreadICsInText*.

simpl.

auto.

split.

apply *H*.

simpl.

apply *threadOffICListInText*.

apply *H3*.

apply *nth_In*.

apply *H*.

split.

rewrite ← *updateMaintainsListLength* with $(t := (threadOffCPU\ (getThread\ vr\ tl)))$.

apply *H4*.

simpl.

apply *threadOnICListInText*.

auto.

simpl in *H2*.

simpl in *H3*.

simpl in *inThread*.

apply *updateThreadICL*.

split.

simpl.

unfold *AreAllThreadICsInText*.

```
simpl.
apply H3.
split.
apply H4.
simpl.
split.
unfold ICInText.
simpl.
split.
unfold textMappedLinear in tml.
simpl in tml.
assert (cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
        vlookup PC
          (C MMU DS (val r) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) =
        jmp).
apply tml.
auto.
destruct H0.
auto.
destruct H as [H H5].
contradict H0.
unfold vlookup.
simpl.
unfold vaLookup in H5.
unfold vLookup.
rewrite → H5.
discriminate.
apply H2.
apply H3.
apply inThread.
simpl in H2.
```

simpl in *H3*.

simpl in *inThread*.

simpl.

apply *updateThreadICL*.

split.

simpl.

unfold *AreAllThreadICsInText*.

simpl.

apply *H3*.

split.

apply *H4*.

simpl.

apply *popInText*.

apply *H3*.

auto.

auto.

simpl in *H2*.

simpl in *H3*.

simpl in *cfg*.

unfold *pushSIC*.

apply *updateThreadICL*.

simpl.

split.

unfold *AreAllThreadICsInText*.

simpl.

auto.

split.

apply *H4*.

apply *H3*.

apply *nth_In*.

apply *H4*.

simpl.

unfold *popSIC*.

apply *updateThreadICL*.

split.

simpl.

unfold *AreAllThreadICsInText*.

simpl.

auto.

split.

apply *H4*.

simpl.

split.

apply *ICListInTextImpliesICInText* with (*icl* := (*getThreadSICList tid tl*)).

apply *sic*.

apply *inThread*.

apply *nth_In*.

apply *H*.

apply *popInText*.

apply *H3*.

simpl.

auto.

unfold *pushIC*.

simpl.

apply *updateThreadICL*.

split.

unfold *AreAllThreadICsInText*.

simpl.

simpl in *H2*.

unfold *validThreadIDs* in *H4*.

apply *H3*.

split.

apply *H4*.

simpl.

split.

unfold *ICInText*.

simpl.

simpl in *H2*.

simpl in *cfg*.

unfold *validCFG* in *cfg*.

simpl in *cfg*.

split.

destruct *H* as [*canExec H*].

destruct *H* as [*inst H*].

apply *cfg23* in *cfg*.

unfold *validCFG3* in *cfg*.

apply *cfg*.

auto.

apply *cfg23* in *cfg*.

apply *cfg*.

apply *H*.

unfold *AreAllThreadICsInText* in *H3*.

simpl in *H3*.

unfold *getThreadICList*.

assert (*In* (*getThread tid tl*) *tl*).

apply *nth_In*.

apply *H4*.

apply *popInText*.

auto.

Qed.

Theorem *staySICInText*: ∀ (*c1 c2* : *config*),

(*c1* ==> *c2*) ∧

(*pcInText c1*) ∧

(*AreAllThreadICsInText* (*getThreadList c1*) (*c1*)) ∧

(*validThreadIDs c1*) ∧

(*validCFG c1* (*getCFG c1*)) ∧

(*In* (*getThread* (*getCurrThread c1*) (*getThreadList c1*)) (*getThreadList c1*)) ∧

(*textMappedLinear c1*) ∧

(*AreAllThreadSICsInText* (*getThreadList c1*) (*c1*))

→

(*AreAllThreadSICsInText* (*getThreadList c2*) (*c2*)).

```
Proof.
intros.
destruct H as [H1 H2].
destruct H2 as [H2 H3].
destruct H3 as [H3 H4].
destruct H4 as [H4 cfg].
destruct cfg as [cfg inThread].
destruct inThread as [inThread tml].
destruct tml as [tml sic].
destruct H1.
destruct c.
auto.
auto.
auto.
auto.
auto.
auto.
simpl.
unfold AreAllThreadSICsInText.
simpl.
simpl in H3.
simpl in H2.
simpl in inThread.
```

simpl in *tml*.

simpl in *sic*.

simpl in *cfg*.

unfold *AreAllThreadSICsInText* in *sic*.

simpl in *sic*.

intros.

apply *updateMMUICListInText*.

split.

apply *goodSICImpliesOut* with (*cs* := *cs*) (*ce* := *ce*) (*mmu* := *MMU*).

split.

apply *sic*.

auto.

destruct *H* as [*I1* *H*].

destruct *H* as [*I2* *H*].

destruct *H* as [*I3* *H*].

destruct *H* as [*I4* *H*].

destruct *H* as [*I5* *H*].

rewrite → *I3* in *I5*.

auto.

apply *sic*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

simpl.

simpl in *H3*.

apply *updateThreadSICL*.

split.

simpl.

unfold *AreAllThreadSICsInText*.

simpl.

unfold *AreAllThreadSICsInText* in *H3*.

simpl in *H3*.

apply *sic*.

split.

destruct *H* as [*H H1*].

destruct *H1* as [*H1 H5*].

destruct *H4* as [*H4 H6*].

auto.

simpl.

auto.

simpl in *H2*.

simpl in *H3*.

simpl in *inThread*.

simpl in *cfg*.

apply *updateThreadSICL*.

split.

simpl.

apply *updateThreadSICL*.

split.

simpl.

unfold *AreAllThreadSICsInText*.

simpl.

auto.

split.

apply *H*.

simpl.

apply *threadOffSICListInText*.

apply *sic*.

apply *nth_In*.

apply *H*.

split.

rewrite ← *updateMaintainsListLength* with (t := (*threadOffCPU* (*getThread vr tl*))).

apply *H4*.

simpl.

apply *threadOnSICListInText*.

auto.

simpl in *H2*.

simpl in *H3*.

simpl in *inThread*.

apply *updateThreadSICL*.

split.

simpl.

unfold *AreAllThreadSICsInText*.

simpl.

apply *sic*.

split.

apply *H4*.

simpl.

apply *sic*.

apply *inThread*.

simpl in *H2*.

simpl in *H3*.

simpl in *inThread*.

simpl.

apply *updateThreadSICL*.

split.

simpl.

unfold *AreAllThreadSICsInText*.

simpl.

apply *sic*.

split.

apply *H4*.

simpl.

apply *sic*.

auto.

auto.

simpl.

unfold *pushSIC*.

apply *updateThreadSICL*.

split.

simpl.

unfold *AreAllThreadSICsInText*.

simpl.

auto.

split.

apply *H4*.

simpl.

split.

apply *ICListInTextImpliesICInText* with (*icl* := (*getThreadICList tid tl*)).

apply *H3*.

apply *inThread*.

apply *nth_In*.

apply *H*.

apply *sic*.

auto.

unfold *popSIC*.

simpl.

apply *updateThreadSICL*.

split.

simpl.

```
unfold AreAllThreadSICsInText.
simpl.
auto.
split.
apply H4.
simpl.
apply popInText.
apply sic.
auto.
unfold pushIC.
simpl.
apply updateThreadSICL.
split.
unfold AreAllThreadSICsInText.
simpl.
apply sic.
split.
apply H4.
simpl.
apply sic.
apply inThread.
Qed.
```

B.12 InvProofs.v

```
Require Import Arith.
Require Import List.
Require Import Coq.Logic.Classical_Prop.
Require Export Semantics.
Require Export ThreadProofs.
Theorem alwaysGoodTH : ∀ (c1 c2 : config),
```

$(c1 \implies c2) \wedge$

$(In\ (getTH\ c1)\ (getCFG\ c1)) \rightarrow$

$(In\ (getTH\ c2)\ (getCFG\ c2))$.

```
Proof.
intros.
destruct H as [H1 H2].
destruct H1.
destruct c.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
simpl.
apply H.
auto.
auto.
auto.
Qed.
```

B.13 ThreadProofs.v

```
Require Import Arith.

Require Import List.

Require Import Coq.Logic.Classical_Prop.

Require Export Semantics.

Theorem updateEmptyThreadList: ∀ (tid : nat) (t : SVAThread),

updateThread tid t nil = nil.

Proof.

intros.

destruct tid.

auto.

unfold updateThread.

auto.

Qed.

Theorem updateDifferentThread : ∀ (tid ntid : nat)
                                  (t : SVAThread)
                                  (tl : ThreadList),

tid ≠ ntid → (getThread tid tl) = (getThread tid (updateThread ntid t tl)).

Proof.

intros.

generalize dependent tid.

generalize dependent ntid.

induction tl.

intros.

destruct ntid.

auto.

auto.

intros.

destruct tid.

destruct ntid.
```

contradiction H.

`auto.`

`simpl.`

`unfold` *getThread.*

`auto.`

`unfold` *getThread.*

`simpl.`

`destruct` *ntid.*

`simpl.`

`auto.`

`simpl.`

`apply` *IHtl.*

contradict H.

`rewrite` $\rightarrow$ *H.*

`auto.`

`Qed.`

`Theorem` *addValidThread* : $\forall$ (*t* : *SVAThread*)
(*tl* : *ThreadList*)
(*cfg* : *list nat*)
(*mmu* : *MMU*)
(*ds* : *store*),

(*validThread t cfg mmu ds*) $\wedge$
(*validThreadList tl cfg mmu ds*) $\rightarrow$ *validThreadList* (*t* :: *tl*) *cfg mmu ds.*

`Proof.`

`intros.`

`destruct` *H* `as` [*H1 H2*].

`unfold` *validThreadList.*

`split.`

`auto.`

`auto.`

```
Qed.

Theorem replaceValidThread : ∀ (t : SVAThread)
                                (tl : ThreadList)
                                (cfg : list nat)
                                (mmu : MMU)
                                (ds : store)
                                (n : nat),
(validThread t cfg mmu ds) ∧ (validThreadList tl cfg mmu ds)
→
validThreadList (updateThread n t tl) cfg mmu ds.
Proof.
intros t tl mmu cfg ds.
induction tl.

intros.
destruct H as [H1 H2].
induction n.
auto.
auto.

intros.
destruct H as [H1 H2].
destruct H2 as [H2 H3].
fold validThreadList in H3.

assert (validThreadList (a :: tl) mmu cfg ds).
apply addValidThread.
auto.

destruct n.
simpl.
auto.

unfold updateThread.
simpl.
```

```
fold updateThread.
auto.
Qed.

Theorem validThreadMMU : ∀ (t : SVAThread)
                           (c1 : config)
                           (tlb : TLBTy)
                           (v p : nat),
(validConfig c1) ∧
(validThread t (getCFG c1) (getCMMU c1) (getStore c1)) ∧
(((canThreadSwap t) = true) → (v ≠ (getThreadPC t)) ∧ (v ≠ ((getThreadPC t) - 1)))
→
(validThread t (getCFG c1) (updateTLB v tlb (getCMMU c1)) (getStore c1)).
Proof.
intros.
destruct c1.
simpl.
simpl in H.
destruct H as [H1 H].
destruct H as [H2 H].
destruct t.
destruct b.
simpl in H.
unfold validThread in H2.
unfold validThread.
destruct H as [H3 H4].
auto.
assert ((vLookup (n9 - 1) (updateTLB v tlb m) s) = (vLookup (n9 - 1) m s)).
rewrite ← sameMMURead.
auto.
auto.
rewrite → H.
```

```
auto.

auto.

Qed.

Theorem validThreadStore : ∀ (t : SVAThread) (c : config) (v : nat) (n : tm),

((validConfig c) ∧

(validThread t (getCFG c) (getCMMU c) (getStore c)) ∧

(threadInText t (getCFG c) (getCMMU c) (getTextStart c) (getTextEnd c)) ∧

(((canThreadSwap t) = true) →

  (v ≠ (getThreadPC t)) ∧

  (v ≠ ((getThreadPC t) - 1)) ∧

  (not (In v (getCFG c))))) ∧

(textMappedOnce c)

→

(validThread t (getCFG c) (getCMMU c) (replace (getPhysical (getTLB v (getCMMU c))) n (getStore

c))).

Proof.

intros.

destruct c.

simpl.

simpl in H.

destruct t.

destruct b.

simpl in H.

destruct H as [H H1].

destruct H as [H H2].

destruct H2 as [H3 H2].

destruct H2 as [H4 H2].

destruct H4 as [H10 H11].

unfold validThread.

destruct H3 as [H5 | H6].

destruct H11 as [H12 | H13].
```

left.

`auto`.

left.

`auto`.

`destruct` *H11* `as` [*H12* | *H13*].

left.

`auto`.

right.

`unfold` *vLookup* `in` *H6*.

`unfold` *vLookup*.

`rewrite` ← *sameRead*.

`auto`.

`apply` *H1*.

`rewrite` ← *pred_of_minus*.

`split`.

`auto`.

`destruct` *H2* `as` [*H8 H2*].

`auto`.

`rewrite` ← *pred_of_minus* `in` *H2*.

`apply` *not_eq_sym*.

`apply` *H2*.

`unfold` *validThread*.

`auto`.

`Qed`.

`Theorem` *threadsAlwaysValid* : ∀ (*c1 c2* : *config*),

(*validCFG c1* (*getCFG c1*)) ∧ (*validConfig c1*) ∧ (*textNotWriteable c1*) ∧

(*validThreadList* (*getThreadList c1*) (*getCFG c1*) (*getCMMU c1*) (*getStore c1*)) ∧

(*getTextStart c1*) ≤ (*getPhysical* (*getTLB* (*getPC c1*) (*getCMMU c1*))) ≤ (*getTextEnd c1*) ∧

(*threadListInText* (*getThreadList c1*)

(*getCFG c1*)(*getCMMU c1*) (*getTextStart c1*) (*getTextEnd c1*)) ∧

(*textMappedOnce c1*) ∧

(*c1 ==> c2*)

→

(*validThreadList* (*getThreadList c2*) (*getCFG c2*) (*getCMMU c2*) (*getStore c2*)).

`Proof.`

`intros.`

`destruct` *H* `as` [*vcfg H*].

`destruct` *H* `as` [*H1 H2*].

`destruct` *H2* `as` [*H2 H3*].

`destruct` *H3* `as` [*H4 H3*].

`destruct` *H3* `as` [*H9 H3*].

`destruct` *H3* `as` [*HA H3*].

`destruct` *H3* `as` [*HE H3*].

`destruct` *H3*.

`destruct` *c*.

`auto.`

`auto.`

`simpl.`

`simpl in` *H1*.

`simpl in` *H2*.

`simpl in` *H4*.

`simpl in` *H9*.

`simpl in` *HA*.

`destruct` *H* `as` [*H5 H6*].

`destruct` *H6* `as` [*H6 H7*].

`destruct` *H7* `as` [*H7 H8*].

`induction` *tl*.

`auto.`

`destruct` *HA* `as` [*HA HB*].

fold threadListInText `in` *HB*.

`unfold` *validThreadList*.

`split.`

```
destruct a.
destruct b.
unfold threadInText in HA.
destruct HA as [HA HC].
destruct HC as [HC | HC].
unfold validThread.
left.
auto.
assert (n ≠ n0).
contradict HA.
rewrite ← HA.
contradict HA.
assert (~ canWrite n MMU).
apply H2.
auto.
contradiction.
assert (CFG = (getCFG (C MMU DS Reg PC SP CFG cs ce st asid tid ntid
          ((Thread true n0 i i0) :: tl) gvs gve gl th))).
auto.
assert (MMU = (getCMMU (C MMU DS Reg PC SP CFG cs ce st asid tid ntid
          ((Thread true n0 i i0) :: tl) gvs gve gl th))).
auto.
assert (DS = (getStore (C MMU DS Reg PC SP CFG cs ce st asid tid ntid
          (Thread true n0 i i0 :: tl) gvs gve gl th))).
auto.
rewrite → H0.
rewrite → H3.
rewrite → H10.
apply validThreadStore.
simpl.
split.
```

split.

auto.

split.

unfold *validThreadList* in *H4*.

destruct *H4*.

unfold *validThread* in *H4*.

apply *H4*.

split.

auto.

split.

auto.

split.

contradict H7.

rewrite $\rightarrow$ *H7*.

apply *H2*.

rewrite $\leftarrow$ *pred_of_minus*.

unfold *threadInText* in *HA*.

destruct *HA* as [*HA HD*].

apply *HC*.

simpl in *IHtl*.

simpl in *vcfg*.

unfold *validCFG* in *vcfg*.

apply *cfg23* in *vcfg*.

simpl in *vcfg*.

unfold *validCFG3* in *vcfg*.

assert $((In\ n\ CFG) \rightarrow (cs \leq getPhysical\ (getTLB\ n\ MMU) \leq ce))$.

apply *vcfg*.

contradict H12.

assert $(cs \leq getPhysical\ (getTLB\ n\ MMU) \leq ce)$.

apply *vcfg*.

auto.

assert (*not* (*canWrite n MMU*)).

apply *H2*.

auto.

contradiction.

auto.

auto.

fold validThreadList.

apply *IHtl*.

auto.

unfold *validThreadList* in *H4*.

destruct *H4*.

fold validThreadList in *H0*.

auto.

auto.

auto.

auto.

auto.

auto.

simpl.

simpl in *H1*.

simpl in *H2*.

simpl in *H4*.

simpl in *H9*.

simpl in *HA*.

destruct *H* as [*H10 H*].

destruct *H* as [*H11 H*].

destruct *H* as [*H12 H*].

destruct *H* as [*H13 H*].

destruct *H* as [*H14 H*].

destruct *H* as [*H15 H*].

destruct *H* as [*H16 H*].

induction *tl*.

auto.

unfold *validThreadList*.

split.

unfold *validThreadList* in *H4*.

destruct *H4*.

fold validThreadList in *H3*.

destruct *a*.

destruct *b*.

assert $(n \neq v)$.

unfold *threadListInText* in *HA*.

destruct *HA* as [*HB HA*].

fold threadListInText in *HA*.

unfold *threadInText* in *HB*.

simpl in *HB*.

destruct *HB* as [*HB HD*].

rewrite $\rightarrow$ *H12* in *H14*.

contradict HB.

rewrite $\rightarrow$ *HB*.

contradict HB.

destruct *HB*.

destruct *H14*.

contradict H4.

apply *lt_not_le*.

auto.

contradict H5.

apply *lt_not_le*.

auto.

unfold *threadListInText* in *HA*.

destruct *HA* as [*HB HA*].

fold threadListInText `in` *HA*.

`unfold` *threadInText* `in` *HB*.

`destruct` *HB* `as` [*HB HC*].

`destruct` *HC* `as` [*valid1* | *valid2*].

`unfold` *validThread*.

left.

`auto`.

`assert` (*v* ≠ (*pred n*)).

`destruct` *HB* `as` [*HD HB*].

`rewrite` → *H12* `in` *H14*.

contradict valid2.

`rewrite` ← *valid2*.

contradict valid2.

`destruct` *valid2*.

`destruct` *H14*.

contradict H7.

`apply` *le_not_lt*.

`auto`.

contradict H6.

`apply` *lt_not_le*.

`auto`.

`assert` (*CFG* = (*getCFG* (*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*))).

`auto`.

`assert` (*MMU* = (*getCMMU* (*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*))).

`auto`.

`assert` (*DS* = (*getStore* (*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*))).

`auto`.

`rewrite` → *H6*.

`rewrite` → *H7*.

`rewrite` → *H8*.

`apply` *validThreadMMU*.

auto.

simpl.

split.

apply *H1*.

unfold *validThread* in *H0*.

split.

apply *H0*.

intro.

split.

apply *not_eq_sym*.

apply *H4*.

rewrite ← *pred_of_minus*.

auto.

auto.

fold validThreadList.

apply *IHtl*.

auto.

destruct *H4*.

fold validThreadList in *H3*.

auto.

destruct *HA*.

fold threadListInText in *H3*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

```
simpl.
apply replaceValidThread.
simpl in H1.
simpl in H2.
simpl in H4.
destruct H as [H5 H6].
destruct H6 as [H6 H7].
destruct H7 as [H7 H8].
split.
unfold validThread.
auto.
auto.
simpl.
apply replaceValidThread.
split.
destruct getThread.
destruct getThread.
unfold threadOnCPU.
unfold validThread.
right.
simpl.
rewrite ← minus_n_O.
unfold vLookup.
unfold vaLookup in H.
destruct H as [H17 H].
destruct H as [H18 H].
auto.
apply replaceValidThread.
split.
destruct getThread.
unfold threadOffCPU.
```

unfold *validThread*.

auto.

auto.

simpl.

simpl in *HA*.

apply *replaceValidThread*.

split.

unfold *validThread*.

auto.

auto.

simpl.

simpl in *HA*.

simpl in *H4*.

simpl in *H9*.

simpl in *H1*.

apply *replaceValidThread*.

split.

unfold *validThread*.

auto.

auto.

auto.

simpl.

unfold *pushSIC*.

apply *replaceValidThread*.

split.

unfold *validThread*.

auto.

auto.

simpl.

unfold *popSIC*.

apply *replaceValidThread*.

split.

simpl.

auto.

auto.

simpl.

unfold *pushIC*.

apply *replaceValidThread*.

split.

unfold *validThread*.

auto.

auto.

Qed.

Theorem *getThreadExtraList*: $\forall$ (n : *nat*) (t : *SVAThread*) (tl : *ThreadList*),

(*getThread* (*S* *n*) (t :: tl)) = (*getThread* *n* tl).

Proof.

intros.

unfold *getThread*.

simpl.

auto.

Qed.

Theorem *getThreadImpliesIn*: $\forall$ (n : *nat*) (t : *SVAThread*) (tl : *ThreadList*),

In (*getThread* *n* tl) tl $\rightarrow$ *In* (*getThread* *n* tl) (t :: tl).

Proof.

intros.

generalize dependent t.

generalize dependent n.

induction tl.

contradiction.

intros.

```
destruct n.
unfold getThread.
simpl.
auto.
unfold getThread.
simpl.
unfold getThread in H.
simpl in H.
destruct H.
right.
left.
auto.
right.
right.
auto.
Qed.

Theorem updateThreadStillThere: ∀ (t : SVAThread)
                                  (tl : ThreadList)
                                  (tid ntid : nat),
(In (getThread tid tl) tl) ∧ (tid < length tl) →
(In (getThread tid (updateThread ntid t tl)) (updateThread ntid t tl)).
Proof.
intros.
destruct H as [H1 H2].
assert (tid < length (updateThread ntid t tl)).
rewrite ← updateMaintainsListLength.
auto.
apply nth_In.
auto.
Qed.
```

```
Theorem threadIDsAlwaysValid: ∀ (c1 c2 : config),
(c1 ==> c2) ∧ (validThreadIDs c1) → (validThreadIDs c2).
Proof.
intros.
destruct H as [H1 H2].
destruct H1.
destruct c.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
simpl.
simpl in H2.
rewrite ← updateMaintainsListLength.
split.
apply H2.
apply H.
simpl.
rewrite ← updateMaintainsListLength.
rewrite ← updateMaintainsListLength.
split.
apply H.
```

```
apply H2.

simpl.

rewrite ← updateMaintainsListLength.

auto.

simpl.

rewrite ← updateMaintainsListLength.

auto.

auto.

simpl.

unfold pushSIC.

rewrite ← updateMaintainsListLength.

auto.

simpl.

unfold popSIC.

rewrite ← updateMaintainsListLength.

auto.

simpl.

unfold pushIC.

rewrite ← updateMaintainsListLength.

auto.

Qed.

Theorem threadAlwaysThere: ∀ (c1 c2 : config),

(c1 ==> c2) ∧

(validThreadIDs c1) ∧

(In (getThread (getCurrThread c1) (getThreadList c1)) (getThreadList c1))

→

(In (getThread (getCurrThread c2) (getThreadList c2)) (getThreadList c2)).

Proof.

intros.

destruct H as [H1 H2].
```

```
destruct H2 as [H3 H2].
destruct H1.
destruct c.
simpl in H2.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
auto.
simpl.
simpl in H2.
apply updateThreadStillThere.
split.
auto.
apply H3.
auto.
simpl.
simpl in H2.
apply updateThreadStillThere.
split.
apply updateThreadStillThere.
split.
apply nth_In.
```

apply *H*.

apply *H*.

rewrite ← *updateMaintainsListLength*.

apply *H*.

simpl.

simpl in *H2*.

apply *updateThreadStillThere*.

split.

auto.

apply *H3*.

simpl.

simpl in *H2*.

apply *updateThreadStillThere*.

split.

auto.

apply *H3*.

auto.

unfold *pushSIC*.

simpl.

simpl in *H2*.

apply *updateThreadStillThere*.

split.

auto.

apply *H3*.

unfold *popSIC*.

simpl.

simpl in *H2*.

apply *updateThreadStillThere*.

split.

auto.

apply *H3*.

```
unfold pushSIC.
simpl.
simpl in H2.
apply updateThreadStillThere.
split.
auto.
apply H3.
Qed.
```

B.14 ThreadTextProofs.v

```
Require Import Arith.
Require Import List.
Require Import Coq.Logic.Classical_Prop.
Require Export Semantics.
Require Export InvProofs.
Require Export ICProofs.
Theorem addThreadInText : ∀ (t : SVAThread)
                            (tl : ThreadList)
                            (cfg : list nat)
                            (mmu : MMU)
                            (cs ce : nat),
(threadInText t cfg mmu cs ce) ∧
(threadListInText tl cfg mmu cs ce) → threadListInText (t :: tl) cfg mmu cs ce.
Proof.
intros.
destruct H as [H1 H2].
unfold threadListInText.
split.
auto.
auto.
```

```
Qed.

Theorem replaceThreadInText : ∀ (t : SVAThread)
                                (tl : ThreadList)
                                (cfg : list nat)
                                (mmu : MMU)
                                (cs ce n : nat),
(threadInText t cfg mmu cs ce) ∧ (threadListInText tl cfg mmu cs ce)
→
threadListInText (updateThread n t tl) cfg mmu cs ce.
Proof.
intros t tl cfg mmu cs ce.
induction tl.

intros.
destruct H as [H1 H2].
induction n.
auto.
auto.

intros.
destruct H as [H1 H2].
destruct H2 as [H2 H3].
fold threadListInText in H3.

assert (threadListInText (a :: tl) cfg mmu cs ce).
apply addThreadInText.
auto.

destruct n.
simpl.
auto.

unfold updateThread.
simpl.
fold updateThread.
```

```
auto.
Qed.
Theorem textInThreadMMU : ∀ (t : SVAThread)
                            (cfg : list nat)
                            (tlb : TLBTy)
                            (mmu : MMU)
                            (v cs ce : nat),
(threadInText t cfg mmu cs ce) ∧
(((canThreadSwap t) = true) →
  (v ≠ (getThreadPC t)) ∧
  (not (In (getThreadPC t) cfg) → v ≠ ((getThreadPC t) - 1)))
→
(threadInText t cfg (updateTLB v tlb mmu) cs ce).
Proof.
intros.
destruct H as [H1 H2].
destruct t.
destruct b.
unfold canThreadSwap in H2.
destruct H2.
auto.
simpl in H0.
unfold threadInText.
rewrite ← sameMMULookup.
split.
apply H1.
unfold threadInText in H1.
destruct H1 as [H1 H2].
apply imply_to_or in H0.
destruct H2 as [H2 | H2].
left.
```

```
auto.
destruct H0 as [H0 | H0].
apply NNPP in H0.
left.
auto.
right.
rewrite ← sameMMULookup.
auto.
rewrite ← pred_of_minus in H0.
apply not_eq_sym.
apply H0.
unfold getThreadPC in H.
apply not_eq_sym.
auto.
auto.
Qed.

Theorem TLAlwaysInText: ∀ (c1 c2 : config),
(c1 ==> c2) ∧
(threadListInText
   (getThreadList c1) (getCFG c1) (getCMMU c1) (getTextStart c1) (getTextEnd c1)) ∧
(validCFG c1 (getCFG c1)) ∧
(textMappedLinear c1) ∧
(pcInText c1)
→
(threadListInText
   (getThreadList c2) (getCFG c2) (getCMMU c2) (getTextStart c2) (getTextEnd c2)).
Proof.
intros.
destruct H as [H1 H2].
destruct H2 as [H2 cfg].
destruct cfg as [cfg line].
```

```
destruct line as [line pct].
destruct H1.
destruct c.
auto.
auto.
auto.
auto.
auto.
auto.
simpl.
simpl in H2.
destruct H as [H10 H].
destruct H as [H11 H].
destruct H as [H12 H].
destruct H as [H13 H].
destruct H as [H14 H].
destruct H as [H15 H].
destruct H as [H16 H].
induction tl.
auto.
simpl in cfg.
simpl in IHtl.
unfold threadListInText.
split.
unfold threadListInText in H2.
destruct H2 as [H2 H3].
fold threadListInText in H3.
apply textInThreadMMU.
split.
auto.
```

destruct *a*.

destruct *b*.

simpl.

intros.

destruct *H0*.

unfold *threadInText* in *H2*.

destruct *H2* as [*H4* *H5*].

rewrite → *H12* in *H14*.

split.

contradict H14.

rewrite → *H14*.

apply *and_not_or*.

split.

apply *le_not_lt*.

apply *H4*.

apply *le_not_lt*.

apply *H4*.

rewrite ← *pred_of_minus*.

apply *or_to_imply*.

destruct *H5*.

left.

contradict H0.

auto.

right.

contradict H0.

rewrite ← *H0*.

contradict H0.

destruct *H14* as [*H14a* | *H14b*].

destruct *H0*.

apply *le_not_lt* in *H0*.

contradiction.

destruct *H0*.

apply *le_not_lt* in *H1*.

contradiction.

simpl.

intros.

contradict H0.

auto.

fold threadListInText.

apply *IHtl*.

destruct *H2* as [*H2 H20*].

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

simpl.

simpl in *H2*.

apply *replaceThreadInText*.

split.

simpl.

split.

simpl in *cfg*.

unfold *validCFG* in *cfg*.

simpl in *cfg*.

assert (*validCFG3 CFG MMU cs ce*).

apply *cfg23*.

```
auto.
unfold validCFG3 in H0.
apply H0.
apply H.
left.
apply H.
auto.
simpl.
simpl in H2.
apply replaceThreadInText.
split.
unfold textMappedLinear in line.
simpl in line.
unfold pcInText in pct.
simpl in pct.
assert ((cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce) ∨
vlookup PC
        (C MMU DS (val vr) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) =
    jmp).
apply line.
auto.
destruct H0.
destruct getThread.
destruct getThread.
simpl.
split.
auto.
right.
auto.
destruct H as [exec swap].
destruct swap as [swap canswap].
```

unfold *vlookup* in *H0*.

simpl in *H0*.

unfold *vaLookup* in *swap*.

unfold *vLookup* in *H0*.

contradict H0.

rewrite → *swap*.

discriminate.

apply *replaceThreadInText*.

split.

destruct *getThread*.

unfold *threadOffCPU*.

simpl.

auto.

auto.

simpl.

apply *replaceThreadInText*.

split.

simpl.

auto.

auto.

simpl.

apply *replaceThreadInText*.

split.

simpl.

auto.

auto.

auto.

simpl.

unfold *pushSIC*.

apply *replaceThreadInText*.

split.

simpl.

auto.

auto.

simpl.

unfold *popSIC*.

apply *replaceThreadInText*.

split.

simpl.

auto.

auto.

simpl.

unfold *pushIC*.

apply *replaceThreadInText*.

split.

simpl.

auto.

auto.

Qed.

Theorem *stayLinear*: $\forall$ (*c1 c2* : *config*),

(*c1* ==> *c2*) $\wedge$

(*textMappedLinear c1*) $\wedge$

(*pcInText c1*) $\wedge$

(*validConfig c1*) $\wedge$

(*textNotWriteable c1*) $\wedge$

(*textMappedOnce c1*) $\wedge$

(*validCFG c1* (*getCFG c1*))

$\rightarrow$

(*textMappedLinear c2*).

Proof.

intros.

destruct *H* as [*H1 H2*].

destruct *H2* as [*H2 H3*].

destruct *H3* as [*H3 H4*].

destruct *H4* as [*H4 H5*].

destruct *H5* as [*H5 H6*].

destruct *H6* as [*H6 H7*].

destruct *H1*.

destruct *c*.

auto.

auto.

unfold *textMappedLinear*.

simpl.

intros.

destruct *H* as [*HA H*].

destruct *H* as [*HB H*].

destruct *H* as [*HC H*].

unfold *textMappedLinear* in *H2*.

simpl in *H2*.

assert ($cs \leq getPhysical\ (getTLB\ (S\ v)\ MMU) \leq ce \vee$

$vlookup\ v$

$(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) = jmp$).

apply *H2*.

auto.

destruct *H1* as [*H1* | *H1*].

left.

auto.

right.

assert ($n \neq v$).

unfold *textNotWriteable* in *H5*.

assert ($not\ (canWrite\ v\ MMU)$).

apply *H5*.

auto.

contradict HC.

rewrite → *HC*.

auto.

unfold *vlookup*.

unfold *vLookup*.

simpl.

rewrite ← *sameRead*.

unfold *vlookup* in *H1*.

unfold *vLookup* in *H1*.

auto.

auto.

auto.

auto.

auto.

unfold *textMappedLinear*.

simpl.

intros.

destruct *H* as [*HA H*].

destruct *H* as [*HB H*].

destruct *H* as [*HC H*].

destruct *H* as [*HD H*].

destruct *H* as [*HE H*].

rewrite → *HC* in *HE*.

unfold *textNotWriteable* in *H5*.

unfold *textMappedOnce* in *H6*.

unfold *textMappedLinear* in *H2*.

simpl in *H2*.

assert (*v0* ≠ *v*).

contradict H0.

rewrite → *H0*.

rewrite → *tlbSet*.

contradict HD.

apply *and_not_or*.

split.

apply *le_not_lt*.

apply *HD*.

apply *le_not_lt*.

apply *HD*.

apply *H*.

rewrite ← *sameMMULookup* in *H0*.

assert (*cs* ≤ *getPhysical* (*getTLB* (*S v0*) *MMU*) ≤ *ce* ∨

vlookup v0

(*C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th*) = *jmp*).

apply *H2*.

auto.

destruct *H8* as [*H8* | *H8*].

left.

rewrite ← *sameMMULookup*.

auto.

contradict HE.

rewrite ← *HE*.

apply *and_not_or*.

split.

apply *le_not_lt*.

apply *H8*.

apply *le_not_lt*.

apply *H8*.

right.

unfold *vlookup*.

unfold *vLookup*.

simpl.

unfold *vlookup* in *H8*.

unfold *vLookup* in *H8*.

simpl in *H8*.

rewrite ← *sameMMULookup*.

apply *H8*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

Qed.

Theorem *TLInTextToT*: ∀ (*tl* : *ThreadList*) (*cfg* : *list nat*) (*mmu* : *MMU*) (*cs ce* : *nat*) (*id* : *nat*) , *threadListInText tl cfg mmu cs ce* → *threadInText* (*getThread id tl*) *cfg mmu cs ce*.

Proof.

intros *tl cfg mmu cs ce*.

induction *tl*.

intros.

destruct *id*.

auto.

auto.

```
intros.
induction id.
unfold getThread.
simpl.
unfold threadListInText in H.
destruct H as [H1 H2].
fold threadListInText in H2.
auto.
unfold threadListInText in H.
destruct H as [H1 H2].
fold threadListInText in H2.
apply IHtl.
auto.
Qed.
Theorem stayPCInText: ∀ (c1 c2 : config),
(c1 ==> c2) ∧
(textMappedLinear c1) ∧
(pcInText c1) ∧
(validConfig c1) ∧
(textNotWriteable c1) ∧
(textMappedOnce c1) ∧
(validCFG c1 (getCFG c1)) ∧
(In (getTH c1) (getCFG c1)) ∧
(threadListInText (getThreadList c1)
  (getCFG c1)(getCMMU c1) (getTextStart c1) (getTextEnd c1)) ∧
(AreAllThreadICsInText (getThreadList c1) c1) ∧
(In (getThread (getCurrThread c1) (getThreadList c1)) (getThreadList c1))
→
(pcInText c2).
Proof.
intros.
```

```
destruct H as [H1 H2].
destruct H2 as [H2 H3].
destruct H3 as [H3 H4].
destruct H4 as [H4 H5].
destruct H5 as [H5 H6].
destruct H6 as [H6 H7].
destruct H7 as [H7 TH].
destruct TH as [TH TLT].
destruct TLT as [TLT ICText].
destruct ICText as [ICText goodThreadID].

destruct H1.
destruct c.

unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert (n2 ≤ getPhysical (getTLB (S n0) m) ≤ n3 ∨
      vlookup n0 (C m s t n0 n1 l n2 n3 l0 n4 n5 n6 t0 n7 n8 l1 n9) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold getInsn in H0.
unfold vlookup in H0.
unfold vlookup.
simpl.
unfold vLookup in H0.
unfold vLookup.
simpl in H0.
```

```
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
destruct H0.
unfold vlookup.
simpl.
unfold vaLookup in H0.
unfold vLookup.
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
```

```
destruct H0.
auto.
contradict H0.
destruct H.
destruct H0.
unfold vlookup.
simpl.
unfold vaLookup in H0.
unfold vLookup.
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS (val v) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
rewrite → H0.
discriminate.
unfold pcInText.
```

```
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
      vlookup PC
        (C MMU DS (val v) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold validCFG in H7.
apply cfg23 in H7.
simpl in H7.
unfold validCFG3 in H7.
apply H7.
apply H.
unfold pcInText.
simpl.
simpl in TH.
simpl in TLT.
unfold textMappedLinear in H2.
```

```
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
rewrite ← sameMMULookup.
auto.
destruct H as [HA H].
destruct H as [HB H].
destruct H as [HC H].
destruct H as [HD H].
destruct H as [HE H].
destruct H as [HF H].
rewrite → HC in HE.
contradict HE.
rewrite ← HE.
apply and_not_or.
split.
apply le_not_lt.
apply H0.
apply le_not_lt.
apply H0.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
destruct H0 as [H0 H10].
```

```
rewrite → H0.

discriminate.

unfold pcInText.

simpl.

unfold textMappedLinear in H2.

simpl in H2.

assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨

    vlookup PC

      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).

apply H2.

apply H3.

destruct H0.

auto.

contradict H0.

destruct H.

unfold vlookup.

simpl.

unfold vLookup.

unfold vaLookup in H0.

destruct H0 as [H0 H10].

rewrite → H0.

discriminate.

unfold pcInText.

simpl.

unfold validCFG in H7.

apply cfg23 in H7.

simpl in H7.

unfold validCFG3 in H7.

apply H7.

apply H.

unfold pcInText.
```

```
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
destruct H0 as [H0 H10].
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold validCFG in H7.
apply cfg23 in H7.
simpl in H7.
unfold validCFG3 in H7.
apply H7.
apply H.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
```

```
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS (val vr) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
destruct H0 as [H0 H10].
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS (val vr) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
```

unfold *vLookup*.

unfold *vaLookup* in *H0*.

rewrite → *H0*.

discriminate.

unfold *pcInText*.

simpl.

unfold *textMappedLinear* in *H2*.

simpl in *H2*.

assert ($cs \leq getPhysical\ (getTLB\ (S\ PC)\ MMU) \leq ce \vee$

vlookup PC

$(C\ MMU\ DS\ Reg\ PC\ SP\ CFG\ cs\ ce\ st\ asid\ tid\ ntid\ tl\ gvs\ gve\ gl\ th) = jmp$).

apply *H2*.

apply *H3*.

destruct *H0*.

auto.

contradict H0.

destruct *H*.

destruct *H0* as [*H0 HA*].

unfold *vlookup*.

simpl.

unfold *vLookup*.

unfold *vaLookup* in *H0*.

rewrite → *H0*.

discriminate.

unfold *pcInText*.

simpl.

simpl in *TLT*.

simpl in *TH*.

assert (*threadInText* (*getThread vr tl*) *CFG MMU cs ce*).

apply *TLInTextToT*.

auto.

destruct *getThread*.

destruct *b*.

unfold *threadInText* in *H0*.

destruct *H0* as [*H0* *H1*].

unfold *getThreadPC*.

auto.

unfold *canThreadSwap* in *H*.

destruct *H*.

destruct *H1*.

destruct *H8* as [*H8* *H100*].

contradict *H8*.

auto.

unfold *pcInText*.

simpl.

unfold *validCFG* in *H7*.

apply *cfg23* in *H7*.

simpl in *H7*.

unfold *validCFG3* in *H7*.

apply *H7*.

apply *TH*.

unfold *pcInText*.

simpl.

destruct *H* as [*H8* *H*].

destruct *H* as [*H9* *H*].

destruct *H* as [*H* *len*].

rewrite → *H*.

unfold *textMappedLinear* in *H2*.

simpl in *H2*.

simpl in *goodThreadID*.

unfold *AreAllThreadICsInText* in *ICText*.

simpl in *ICText*.

```
assert (ICListInText (getThreadICList tid tl) cs ce MMU).
apply ICText.
auto.
unfold itop.
destruct (getThreadICList tid tl).
simpl in len.
contradict len.
apply le_not_lt.
auto.
simpl.
unfold ICListInText in H0.
destruct H0 as [H0 H00].
fold ICListInText in H00.
apply H0.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
      vlookup PC
        (C MMU DS (val vr) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
```

```
destruct H0 as [H0 H10].
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
destruct H0 as [H0 NE].
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS Reg PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
```

```
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
destruct H0 as [H0 sic].
rewrite → H0.
discriminate.
unfold pcInText.
simpl.
unfold textMappedLinear in H2.
simpl in H2.
assert ( cs ≤ getPhysical (getTLB (S PC) MMU) ≤ ce ∨
    vlookup PC
      (C MMU DS (val f) PC SP CFG cs ce st asid tid ntid tl gvs gve gl th) = jmp).
apply H2.
apply H3.
destruct H0.
auto.
contradict H0.
destruct H.
unfold vlookup.
simpl.
unfold vLookup.
unfold vaLookup in H0.
destruct H0 as [H0 H10].
rewrite → H0.
```

```
discriminate.
Qed.
```

B.15 SVAOS.v

```
Require Import Arith.
Require Import List.
Require Import Coq.Logic.Classical_Prop.
Require Import Semantics.
Theorem threadIsValid: ∀ (cfg : list nat) (mmu : MMU) (ds : store) (tl : ThreadList) (n : nat),
validThreadList tl cfg mmu ds → validThread (getThread n tl) cfg mmu ds.
Proof.
intros cfg mmu ds tl.
induction tl.
intros.
unfold getThread.
destruct n.
auto.
auto.
auto.
intros.
inversion H.
destruct n.
unfold getThread.
simpl.
auto.
fold validThreadList in H1.
assert (validThreadList (a :: tl) cfg mmu ds).
unfold validThreadList.
split.
```

```
auto.
fold validThreadList.
auto.
apply IHtl.
auto.
Qed.

Theorem cfisafe : ∀ c1 c2 : config,
  c1 ==> c2 ∧
  (validThreadList (getThreadList c1) (getCFG c1) (getCMMU c1) (getStore c1)) ∧
  (In (getTH c1) (getCFG c1))
  →
  (getPC c2) = (getPC c1 + 1) ∨
  (In (getPC c2) (getCFG c1)) ∨
  ((vlookup (minus (getPC c2) 1) c2) = svaSwap) ∨
  ((getPC c2) = (getICPC (itop (getThreadICList (getCurrThread c1) (getThreadList c1))))).
Proof.
intros c1 c2.
intro H.
destruct H.
destruct H.
destruct c.
left.
simpl.
rewrite → plus_comm.
simpl.
reflexivity.
left.
simpl.
rewrite → plus_comm.
simpl.
reflexivity.
```

left.

simpl.

rewrite → *plus_comm*.

simpl.

reflexivity.

left.

simpl.

rewrite → *plus_comm*.

simpl.

reflexivity.

left.

simpl.

rewrite → *plus_comm*.

simpl.

reflexivity.

right.

left.

simpl.

destruct *H* as [*H1* *H2*].

apply *H2*.

left.

simpl.

rewrite → *plus_comm*.

simpl.

reflexivity.

left.

simpl.

rewrite → *plus_comm*.

simpl.

reflexivity.

right.

left.

`simpl`.

`destruct` *H* `as` [*H1* *H2*].

`apply` *H2*.

`simpl`.

left.

`rewrite` → *plus_comm*.

`auto`.

right.

left.

`simpl`.

`destruct` *H* `as` [*H1* *H2*].

`apply` *H2*.

`simpl`.

left.

`rewrite` → *plus_comm*.

`auto`.

left.

`simpl`.

`rewrite` → *plus_comm*.

`simpl`.

`auto`.

left.

`simpl`.

`rewrite` → *plus_comm*.

`simpl`.

`auto`.

`simpl`.

`simpl in` *H0*.

right.

`unfold` *vlookup*.

`unfold` *vLookup*.

`simpl`.

`destruct` *H0* `as` [*H0 TH*].

`assert` (*validThread* (*getThread vr tl*) *CFG MMU DS*).

`apply` *threadIsValid*.

`auto`.

`destruct` *H*.

`destruct` *H2*.

`destruct` *getThread*.

`unfold` *validThread* `in` *H1*.

`unfold` *canThreadSwap* `in` *H3*.

`destruct` *H3* `as` [*H3 H4*].

`rewrite` → *H3* `in` *H1*.

`simpl`.

`unfold` *vLookup* `in` *H1*.

`destruct` *H1*.

left.

`auto`.

right.

`auto`.

`simpl`.

right.

left.

`simpl in` *H0*.

`apply` *H0*.

`simpl`.

right.

right.

`simpl in` *H0*.

right.

`destruct` *H* `as` [*H1 H2*].

`destruct` *H2* `as` [*H2 H3*].

`destruct` *H3* `as` [*H3 H4*].

`auto`.

`simpl`.

left.

`rewrite` → *plus_comm*.

`auto`.

`simpl`.

left.

`rewrite` → *plus_comm*.

`auto`.

`simpl`.

left.

`rewrite` → *plus_comm*.

`auto`.

`simpl`.

left.

`rewrite` → *plus_comm*.

`auto`.

`Qed`.

`Theorem` *NXText* : ∀ (*c1 c2* : *config*)

(*v* : *nat*),

(*getTextStart c1*) ≤ (*getPhysical* (*getTLB v* (*getCMMU c1*))) ≤ (*getTextEnd c1*) ∧

(*validConfig c1*) ∧

(*textNotWriteable c1*) ∧

(*textMappedOnce c1*) ∧

c1 ==> *c2*

→ (*vLookup v* (*getCMMU c1*) (*getStore c1*)) =

(*vLookup v* (*getCMMU c2*) (*getStore c2*)).

```
Proof.
intros.
destruct H as [h1 H].
destruct H as [h2 H].
destruct H as [h3 H].
destruct H as [h4 H].

destruct H.
destruct c.

auto.

auto.

simpl.
unfold vLookup.
simpl in h1.
simpl in h2.
simpl in h3.
destruct H.
destruct H0.

assert (getPhysical (getTLB v MMU) ≠ getPhysical (getTLB n MMU)).

simpl in h4.
assert (~ canWrite v MMU).
apply h3.
auto.

assert (v ≠ n).
contradict H2.
rewrite → H2.
apply H1.
apply h4.
auto.

rewrite ← sameRead.
auto.
```

apply *H2*.

auto.

auto.

auto.

simpl.

destruct *H* as [*h5* *H*].

destruct *H* as [*h6* *H*].

destruct *H* as [*h7* *H*].

destruct *H* as [*h8* *H*].

destruct *H* as [*h9* *H*].

rewrite → *h7* in *h9*.

simpl in *h1*.

simpl in *h2*.

simpl in *h3*.

simpl in *h4*.

assert (*v* ≠ *v0*).

destruct *h9* as [*ha* | *hb*].

contradict *ha*.

apply *le_not_lt*.

rewrite ← *ha*.

apply *h1*.

contradict *hb*.

apply *le_not_lt*.

rewrite ← *hb*.

apply *h1*.

apply *sameMMURead*.

apply *H0*.

auto.

auto.

auto.

```
auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.
Qed.
```

B.16 Multi.v

```
Require Import Semantics.

Require Import SVAOS.

Require Import List.

Require Import Relations.

Require Import ICText.

Require Import ICProofs.

Require Import ThreadProofs.

Require Import ThreadTextProofs.

Inductive multi {X:Type} (R: relation X) : relation X :=
  | multi_refl : ∀ (x : X), multi R x x
  | multi_step : ∀ (x y z : X),
                    R x y → multi R y z → multi R x z.

Definition multistep := multi step.

Notation " t '==>*' t' " := (multistep t t') (at level 40).
```

```
Theorem TranthreadsAlwaysValid: ∀ (c1 c2 : config),
(validCFG c1 (getCFG c1)) ∧ (validConfig c1) ∧ (textNotWriteable c1) ∧
(validThreadList (getThreadList c1) (getCFG c1) (getCMMU c1) (getStore c1)) ∧
(getTextStart c1) ≤ (getPhysical (getTLB (getPC c1) (getCMMU c1))) ≤ (getTextEnd c1) ∧
(threadListInText (getThreadList c1)
  (getCFG c1)(getCMMU c1) (getTextStart c1) (getTextEnd c1)) ∧
(textMappedOnce c1) ∧
(pcInText c1) ∧
(textMappedLinear c1) ∧
(In (getTH c1) (getCFG c1)) ∧
(AreAllThreadICsInText (getThreadList c1) c1) ∧
(In (getThread (getCurrThread c1) (getThreadList c1)) (getThreadList c1)) ∧
(validThreadIDs c1) ∧
(AreAllThreadSICsInText (getThreadList c1) (c1)) ∧
(c1 ==>* c2)
→
(validThreadList (getThreadList c2) (getCFG c2) (getCMMU c2) (getStore c2)).
Proof.
intros.
destruct H as [vcfg H].
destruct H as [H1 H2].
destruct H2 as [H2 H3].
destruct H3 as [H4 H3].
destruct H3 as [H9 H3].
destruct H3 as [HA H3].
destruct H3 as [HE H3].
destruct H3 as [pc H3].
destruct H3 as [tml H3].
destruct H3 as [inth H3].
destruct H3 as [goodic H3].
destruct H3 as [inThread H3].
```

destruct *H3* as [*validTIDs H3*].

destruct *H3* as [*goodsic H3*].

induction *H3*.

auto.

assert (*validThreadList* (*getThreadList y*) (*getCFG y*) (*getCMMU y*) (*getStore y*)).

apply *threadsAlwaysValid* with (*c1* := *x*).

repeat (split ; auto).

apply *IHmulti*.

unfold *validCFG*.

apply *cfg32*.

apply *alwaysValidCFG* with (*c1* := *x*).

unfold *validCFG* in *vcfg*.

apply *cfg23* in *vcfg*.

auto.

apply *alwaysValid* with (*c1* := *x*).

auto.

apply *neverWriteText* with (*c1* := *x*).

auto.

apply *threadsAlwaysValid* with (*c1* := *x*).

repeat (split ; auto).

apply *stayPCInText* with (*c1* := *x*).

repeat (split ; auto).

apply *TLAlwaysInText* with (*c1* :=*x*).

repeat (split ; auto).

apply *neverMapTextTwice* with (*c1* := *x*).

repeat (split ; auto).

apply *stayPCInText* with (*c1* := *x*).

repeat (split ; auto).

apply *stayLinear* with (*c1* := *x*).

repeat (split ; auto).

apply *alwaysGoodTH* with (*c1* := *x*).

```
repeat (split ; auto).
apply stayICInText with (c1 := x).
repeat (split ; auto).
unfold validThreadList in H4.
repeat (split ; auto).
apply threadAlwaysThere with (c1 := x).
repeat (split ; auto).
apply threadIDsAlwaysValid with (c1 := x).
repeat (split ; auto).
apply staySICInText with (c1 := x).
repeat (split ; auto).
Qed.

Theorem TransalwaysGoodTH: ∀ (c1 c2 : config),
(c1 ==>* c2) ∧
(In (getTH c1) (getCFG c1)) →
(In (getTH c2) (getCFG c2)).
Proof.
intros.
destruct H as [H1 H2].
induction H1.
auto.
apply IHmulti.
apply alwaysGoodTH with (c1 := x).
repeat (split ; auto).
Qed.

Theorem Transcfisafe : ∀ (c1 c2 c3 c4: config),
  (pcInText c1) ∧
  (validThreadIDs c1) ∧
  (validCFG c1 (getCFG c1)) ∧
  (In (getThread (getCurrThread c1) (getThreadList c1)) (getThreadList c1)) ∧
  (textMappedLinear c1) ∧
```

$(AreAllThreadICsInText\ (getThreadList\ c1)\ (c1)) \wedge$

$(AreAllThreadSICsInText\ (getThreadList\ c1)\ (c1)) \wedge$

$(validThreadList\ (getThreadList\ c1)\ (getCFG\ c1)\ (getCMMU\ c1)\ (getStore\ c1)) \wedge$

$(In\ (getTH\ c1)\ (getCFG\ c1)) \wedge$

$(validConfig\ c1) \wedge$

$(textNotWriteable\ c1) \wedge$

$(textMappedOnce\ c1) \wedge$

$(threadListInText\ (getThreadList\ c1)\ (getCFG\ c1)\ (getCMMU\ c1)$

$(getTextStart\ c1)\ (getTextEnd\ c1)) \wedge$

$(c1 ==>^*\ c3) \wedge$

$(c3 ==>\ c4) \wedge$

$(c4 ==>^*\ c2)$

$\rightarrow$

$(getPC\ c4) = (getPC\ c3 + 1) \vee$

$(In\ (getPC\ c4)\ (getCFG\ c3)) \vee$

$((vlookup\ (minus\ (getPC\ c4)\ 1)\ c4) = svaSwap) \vee$

$((getPC\ c4) = (getICPC\ (itop\ (getThreadICList\ (getCurrThread\ c3)\ (getThreadList\ c3)))))).$

```
Proof.
intros.
destruct H as [I1 I].
destruct I as [I2 I].
destruct I as [I3 I].
destruct I as [I4 I].
destruct I as [I5 I].
destruct I as [I6 I].
destruct I as [I7 I].
destruct I as [I8 I].
destruct I as [I9 I].
destruct I as [I10 I].
destruct I as [I13 I].
destruct I as [I14 I].
```

```
destruct I as [I15 I].
destruct I as [I16 I].
destruct I as [I17 I].
apply cfisafe.
split.
auto.
split.
apply TranthreadsAlwaysValid with (c1 := c1).
repeat (split ; auto).
apply TransalwaysGoodTH with (c1 := c1).
repeat (split ; auto).
Qed.

Theorem TranNXText : ∀ (c1 c2 : config) (v : nat),
(getTextStart c1) ≤ (getPhysical (getTLB v (getCMMU c1))) ≤ (getTextEnd c1) ∧
(validConfig c1) ∧
(textNotWriteable c1) ∧
(textMappedOnce c1) ∧
c1 ==>* c2
→ (vLookup v (getCMMU c1) (getStore c1)) =
    (vLookup v (getCMMU c2) (getStore c2)).
Proof.
intros.
destruct H as [H1 H].
destruct H as [H2 H].
destruct H as [H3 H].
destruct H as [H4 H].
induction H.

auto.

assert ((vLookup v (getCMMU x) (getStore x)) = (vLookup v (getCMMU y) (getStore y))).
apply NXText.
auto.
```

rewrite ← *IHmulti*.

apply *H5*.

apply *alwaysInText* with *x*.

auto.

apply *alwaysValid* with *x*.

auto.

apply *neverWriteText* with *x*.

auto.

apply *neverMapTextTwice* with *x*.

auto.

Qed.

B.17 VG.v

Require Import *Arith*.

Require Import *Arith.Lt*.

Require Import *List*.

Require Import *Coq.Logic.Classical_Prop*.

Require Export *Semantics*.

Theorem *ghostNoWrite* : ∀ (*c1 c2* : *config*) (*gv* : *nat*),

(*validConfig c1*) ∧

(*c1* ==> *c2*) ∧

((*getGhostStart c1*) ≤ *gv* ≤ (*getGhostEnd c1*)) ∧

((*In* (*getPhysical* (*getTLB gv* (*getCMMU c1*))) (*getGhost c1*))) ∧

∀ (*v* : *nat*), (*v* < (*getGhostStart c1*) ∨ (*getGhostEnd c1*) < *v*) → (*not* (*In* (*getPhysical* (*getTLB v* (*getCMMU c1*))) (*getGhost c1*))))

→

(*vLookup gv* (*getCMMU c1*) (*getStore c1*)) =

(*vLookup gv* (*getCMMU c2*) (*getStore c2*)).

Proof.

```
intros.
destruct H as [H1 H].
destruct H as [H2 H].
destruct H as [H3 H].
destruct H as [H4 H].
destruct H2.
destruct c.
auto.
auto.
simpl.
simpl in H1.
simpl in H3.
simpl in H4.
simpl in H.
destruct H0.
destruct H2.
destruct H5.
assert (not (In (getPhysical (getTLB n MMU)) gl)).
apply H.
auto.
unfold vLookup.
assert ((getPhysical (getTLB gv MMU)) ≠ (getPhysical (getTLB n MMU))).
contradict H7.
rewrite ← H7.
apply H4.
apply sameRead.
apply H8.
auto.
auto.
auto.
```

simpl.

simpl in *H1*.

simpl in *H3*.

simpl in *H4*.

simpl in *H*.

destruct *H0*.

destruct *H2*.

destruct *H5*.

destruct *H6*.

destruct *H7*.

destruct *H8*.

destruct *H9*.

assert $(v \neq gv)$.

destruct *H8*.

contradict H8.

rewrite $\rightarrow$ *H8*.

apply *le_not_lt*.

apply *H3*.

contradict H8.

rewrite $\rightarrow$ *H8*.

apply *le_not_lt*.

apply *H3*.

apply *sameMMURead*.

auto.

auto.

auto.

auto.

auto.

auto.

auto.

```
auto.

auto.

auto.

auto.

auto.

auto.

auto.

auto.
Qed.

Theorem ghostNoRead : ∀ (c1 c2 : config) (gv : nat), ∃ tv,
  ((validConfig c1) ∧
  (c1 ==> c2) ∧
  ((getGhostStart c1) ≤ gv ≤ (getGhostEnd c1)) ∧
  ((In (getPhysical (getTLB gv (getCMMU c1))) (getGhost c1))) ∧
  ((getGhostStart c1) ≤ tv ≤ (getGhostEnd c1)) ∧
  ((vlookup tv c1) = sec) ∧
  ((getReg c1) ≠ (sec)) ∧
  (∀ (v : nat),
    (v < (getGhostStart c1) ∨ (getGhostEnd c1) < v) →
    (not (In (getPhysical (getTLB v (getCMMU c1))) (getGhost c1))) ∧
    ((vlookup v c1) ≠ (sec))))
  →
  ((getReg c2) ≠ (sec)).
Proof.
intros.
∃ 2.
intros.
destruct H as [H1 H].
destruct H as [H2 H].
destruct H as [H3 H].
```

destruct *H* as [*H4* *H*].

destruct *H* as [*H5* *H*].

destruct *H* as [*H6* *H*].

destruct *H* as [*H7* *H*].

destruct *H2*.

destruct *c*.

simpl.

intro *eq*.

inversion *eq*.

simpl.

simpl in *H*.

simpl in *H1*.

simpl in *H3*.

simpl in *H4*.

unfold *vlookup* in *H*.

simpl in *H*.

destruct *H0*.

destruct *H2*.

destruct *H8*.

destruct *H9*.

rewrite ← *H9*.

apply *H*.

auto.

simpl.

simpl in *H7*.

auto.

simpl.

intro *eq*.

inversion *eq*.

simpl.

intro *eq*.

```
inversion eq.
simpl.
intro eq.
inversion eq.
simpl.
simpl in H7.
auto.
simpl.
simpl in H7.
auto.
simpl.
auto.
simpl.
simpl in H7.
auto.
simpl.
auto.
simpl.
intro eq.
inversion eq.
simpl.
intro eq.
inversion eq.
simpl.
intro eq.
inversion eq.
simpl.
intro eq.
inversion eq.
simpl.
```

intro *eq*.

inversion *eq*.

auto.

auto.

auto.

auto.

auto.

Qed.

References

[1] Apachebench: A complete benchmarking and regression testing suite. http://freshmeat.net/projects/apachebench/, July 2003.

[2] MySQL double free heap corruption vulnerability. `http: //www.securityfocus.com/bid/6718/info`, Jan 2003.

[3] MITKRB5-SA: double free vulnerabilities. `http: //seclists.org/lists/bugtraq/2004/sep/0015.html`, Aug 2004.

[4] *Intel 64 and IA-32 architectures software developer's manual*, volume 3. Intel, 2012.

[5] Amazon web site, 2014. http://www.amazon.com.

[6] iTunes web site, 2014. https://www.apple.com/itunes/.

[7] Linux kernel multiple function remote memory corruption vulnerabilities, March 2014. http://www.securityfocus.com/bid/66279.

[8] Qemu site, 2014. http://www.qemu.org.

[9] TurboTax web site, 2014. https://turbotax.intuit.com/.

[10] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, November 2009.

[11] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *Proc. USENIX Annual Technical Conference*, Atlanta, GA, USA, July 1986.

[12] Advanced Micro Devices. AMD64 architecture programmer's manual volume 2: System programming, September 2006.

[13] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-Level Virtual Instruction Set Architecture. In *Int'l Symp. on Microarchitecture*, December 2003.

[14] Jonathan Afek and Adi Sharabani. Dangling pointer: Smashing the pointer for fun and profit, 2007. http://download.watchfire.com/whitepapers/Dangling-Pointer.pdf.

[15] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society.

[16] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[17] AlephOne. Smashing the stack for fun and profit. http://www.fc.net/phrack/files/p49/p49-14.

[18] Zachary Amsden. Transparent paravirtualization for linux. In *Linux Symposium*, Ottawa, Canada, Jul 2006.

[19] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2013.

[20] Lars Ole Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[21] Apple Computer, Inc. Apple Mac OS X kernel semop local stack-based buffer overflow vulnerability, April 2005. http://www.securityfocus.com/bid/13225.

[22] William A Arbaugh, David J Farber, and Jonathan M Smith. A secure and reliable bootstrap architecture. In *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pages 65–71. IEEE, 1997.

[23] Andrea Arcangeli. Linux kernel mremap local privilege escalation vulnerability, May 2006. http://www.securityfocus.com/bid/18177.

[24] ARM Limited. ARM security technology: Building a secure system using trustzone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009.

[25] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1994.

[26] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[27] Godmar Back and Wilson C. Hsieh. The KaffeOS Java runtime system. *ACM Trans. on Prog. Lang. and Sys.*, 27(4):583–630, 2005.

[28] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, Copper Mountain, CO, USA, 1995.

[29] H. Bos and B. Samwel. Safe kernel programming in the oke. In *Proceedings of IEEE OPENARCH*, 2002.

[30] D. P. Bovet and Marco Cesati. *Understanding the LINUX Kernel*. O'Reilly, Sebastopol, CA, 2^{nd} edition, 2003.

[31] Aaron Brown. *A Decompositional Approach to Computer System Performance*. PhD thesis, Harvard College, April 1997.

[32] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9):36(9):117–126, 1993.

[33] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[34] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, New York, NY, USA, 2009. ACM.

[35] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, New York, NY, USA, 2010. ACM.

[36] Stephen Checkoway and Hovav Shacham. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM.

[37] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium*, pages 177–192, August 2004.

[38] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 2–13, New York, NY, USA, 2008. ACM.

[39] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proc. USENIX Symp. on Op. Sys. Design and Impl.*, pages 179–193, Monterey, CA, USA, November 1994.

[40] B. E. Clark and M. J. Corrigan. Application System/400 performance characteristics. *IBM Systems Journal*, 28(3):407–423, 1989.

[41] Compaq Computer Corporation. *Alpha Architecture Handbook*. Compaq Computer Corporation, 1998.

[42] Kees Cook. Linux kernel CONFIG_HID local memory corruption vulnerability, August 2013. http://www.securityfocus.com/bid/62043.

[43] Kees Cook. Linux kernel CVE-2013-2897 heap buffer overflow vulnerability, August 2013. http://www.securityfocus.com/bid/62044.

[44] corbet. SMP alternatives, December 2005. http://lwn.net/Articles/164121.

[45] Jonathan Corbet. The source of the e1000e corruption bug, October 2008. http://lwn.net/Articles/304105.

[46] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*, volume 2. Intel Corporation, 2002.

[47] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the Thirty-Fifth IEEE Symposium on Security and Privacy*, May 2014. To appear.

[48] John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting applications from hostile operating systems. In *Proceedings of the Nineteenth Internantional Conference on Architectural Support for Programming Languages and Operating Systems*, March 2014.

[49] John Criswell, Nicolas Geoffray, and Vikram Adve. Memory safety for low-level software/hardware interactions. In *Proceedings of the Eighteenth Usenix Security Symposium*, August 2009.

[50] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, Stevenson, WA, USA, October 2007.

[51] John Criswell, Brent Monroe, and Vikram Adve. A virtual instruction set interface for operating system kernels. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, Boston, MA, USA, June 2006.

[52] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Prog. Lang. and Sys.*, pages 13(4):451–490, October 1991.

[53] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 21–35, 1998.

[54] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-world buffer overflow protection for userspace & kernelspace. In *Proceedings of the USENIX Security Symposium*, San Jose, CA, USA, 2008.

[55] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges. In *Proc. Conf. on Code Generation and Optimization*, San Francisco, CA, Mar 2003.

[56] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *International Conference on Software Engineering*, Shanghai, China, May 2006.

[57] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006.

[58] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *ACM Transactions in Embedded Computing Systems (TECS)*, February 2005.

[59] Igor Dobrovitski. Exploit for cvs double free() for linux pserver. `http://seclists.org/lists/bugtraq/ 2003/feb/0042.html`, Feb 2003.

[60] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[61] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proc. Int'l Conf. on Computer Architecture (ISCA)*, pages 26–37, 1997.

[62] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[63] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 2003. Technical Report 2003-1916.

[64] Manuel Fahndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of EuroSys*, 2006.

[65] Ariel J. Feldman, J. Alex Halderman, and Edward W. Felten. Security analysis of the diebold accuvotets voting machine. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, EVT'07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[66] Keir Fraser, Steve Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williams. Safe hardware access with the xen virtual machine monitor. In *Proceedings of the First Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, Boston, MA, USA, October 2004.

[67] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Automatic placement of authorization hooks in the linux security modules framework. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 330–339, New York, NY, USA, 2005. ACM Press.

[68] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001.

[69] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, Berkeley, CA, USA, 2012. USENIX Association.

[70] Michael Golm, Meik Felser, Christian Wawersich, and Jurgen Kleinoder. The JX Operating System. In *Proc. USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, USA, June 2002.

[71] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[72] Georgi Guninski. Linux kernel multiple local vulnerabilities, 2005. http://www.securityfocus.com/bid/11956.

[73] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in haskell. In *Proc. ACM SIGPLAN Int'l Conf. on Functional Programming*, Tallin, Estonia, September 2005.

[74] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference*, June 1998.

[75] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. Inktag: secure applications on an untrusted operating system. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM.

[76] Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker, and Brian Bershad. Language support for extensible operating systems. In *Workshop on Compiler Support for System Software*, Arizona, USA, February 1996.

[77] Galen C. Hunt and James R. Larus. Singularity Design Motivation (Singularity Technical Report 1). Technical Report MSR-TR-2004-105, Microsoft Research, Dec 2004.

[78] Galen C. Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[79] Java Community Process. JSR 121, 2003. http://jcp.org/jsr/detail/121.jsp.

[80] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, 2002.

[81] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[82] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: lightweight kernel protection against returntouser attacks. In *Proceedings of the 21st USENIX conference on Security symposium*, Berkeley, CA, USA, 2012. USENIX Association.

[83] Gerwin Klein et al. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, New York, NY, USA, 2009. ACM.

[84] Joseph Kong. *Designing BSD Rootkits*. No Starch Press, San Francisco, CA, USA, 2007.

[85] Lap Chung Lam and Tzi-cker Chiueh. A general dynamic information flow tracking framework for security applications. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 463–472, Washington, DC, USA, 2006. IEEE Computer Society.

[86] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proc. Conf. on Code Generation and Optimization*, pages 75–88, San Jose, CA, USA, Mar 2004.

[87] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[88] Chris Lattner, Andrew D. Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289, San Diego, CA, USA, June 2007.

[89] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: an organizing principle for recoverable operating systems. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 49–60, New York, NY, USA, 2009. ACM.

[90] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, New York, NY, USA, 2010. ACM.

[91] David Lie, Chandramohan A. Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 178–192, New York, NY, USA, 2003. ACM.

[92] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[93] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[94] LMH. Month of kernel bugs (MoKB) archive, 2006. http://projects.info-pull.com/mokb/.

[95] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter Conference*, 1993.

[96] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[97] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM.

[98] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10. ACM, 2013.

[99] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley Publishing Company, Inc., Redwood City, CA, 1996.

[100] Larry McVoy and Carl Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[101] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: an IDL for hardware programming. In *USENIX Symposium on Operating System Deisgn and Implementation*, San Diego, CA, USA, October 2000.

[102] Sun Microsystems. Sun solaris sysinfo system call kernel memory reading vulnerability, October 2003. http://www.securityfocus.com/bid/8831.

[103] Mindcraft. Webstone: The benchmark for webservers, 2002. http://www.mindcraft.com/webstone.

[104] Brent M. Monroe. Measuring and improving the performance of Linux on a virtual instruction set architecture. Master's thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Urbana, IL, Dec 2005.

[105] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. Rocksalt: better, faster, stronger sfi for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM.

[106] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Trans. on Prog. Lang. and Sys.*, May 1999.

[107] Inc. Motorola. *Programming Environments Manual for 32 Bit Implementations of the PowerPC Architecture*. Motorola, Inc., 2001.

[108] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258, New York, NY, USA, 2009. ACM.

[109] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–344, 1998.

[110] George C. Necula. Proof-carrying code. In *Proc. ACM SIGACT Symp. on Principles of Prog. Lang.*, January 1997.

[111] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. on Prog. Lang. and Sys.*, 2005.

[112] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proc. USENIX Symp. on Op. Sys. Design and Impl.*, 1996.

[113] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 379–394, Washington, DC, USA, 2011. IEEE Computer Society.

[114] Dan R. K. Ports and Tal Garfinkel. Towards application security on untrusted operating systems. In *Proceedings of the 3rd conference on Hot topics in security*, HOTSEC'08, pages 1:1–1:7, Berkeley, CA, USA, 2008. USENIX Association.

[115] Jef Poskanze. thttpd - tiny/turbo/throttling http server, 2000. http://www.acme.com/software/thttpd.

[116] Postmark. Email delivery for web apps, July 2013.

[117] The OpenBSD Project. Openssh, 2006. http://www.openssh.com.

[118] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[119] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[120] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, San Diego, CA, USA, 2004.

[121] Jonathan Salwan and Allan Wirth. http://shell-storm.org/project/ROPgadget.

[122] T. Saulpaugh and C. Mirho. *Inside the JavaOS Operating System.* Addison-Wesley, Reading, MA, USA, 1999.

[123] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2001.

[124] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating System Deisgn and Implementation*, Seattle, WA, USA, October 1996.

[125] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *USENIX Symposium on Operating System Deisgn and Implementation*, pages 213–227, Seattle, WA, October 1996.

[126] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, 2007. ACM.

[127] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, Washington,D.C., August 2001.

[128] J. Shapiro, M. S. Doerrie, E. Northup, S. Sridhar, and M. Miller. Towards a verified, general-purpose operating system kernel. In *1st NICTA Workshop on Operating System Verification*, Sydney, Australia, October 2004.

[129] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the EROS single-level store. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, June 2002.

[130] Amit Singh. *Mac OS X Internals*. Addison-Wesley Professional, 2006.

[131] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 500 Sansome Street, Suite 400, San Francisco CA, 94111, June 2005.

[132] Solar Designer. return-to-libc attack, August 1997. http://www.securityfocus.com/archive/1/7480.

[133] Hannes Frederic Sowa. Linux kernel CVE-2013-4470 multiple local memory corruption vulnerabilities, October 2013. http://www.securityfocus.com/bid/63359.

[134] Paul Starsetz. Linux kernel do_mremap function vma limit local privilege escalation vulnerability, February 2004. http://www.securityfocus.com/bid/9686.

[135] Paul Starzetz. Linux kernel elf core dump local buffer overflow vulnerability. http://www.securityfocus.com/bid/13589.

[136] Paul Starzetz. Linux kernel IGMP multiple vulnerabilities, 2004. http://www.securityfocus.com/bid/11917.

[137] Paul Starzetz and Wojciech Purczynski. Linux kernel setsockopt MCAST_MSFILTER integer overflow vulnerability, 2004. http://www.securityfocus.com/bid/10179.

[138] Paul Starzetz and Wojcieh Purczynski. Linux kernel do_mremap function boundary condition vulnerability, January 2004. http://www.securityfocus.com/bid/9356.

[139] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. ACM SIGACT Symp. on Principles of Prog. Lang.*, 1996.

[140] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proc. USENIX Symp. on Op. Sys. Design and Impl.*, December 2004.

[141] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, 2003.

[142] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst*, 23(1):77–110, 2005.

[143] The Coq Development Team. The Coq proof assistant reference manual (version 8.3), 2010. http://coq.inria.fr/refman/index.html.

[144] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *USENIX Symposium on Operating System Deisgn and Implementation*, pages 75–88, Seattle, WA, USA, November 2006.

[145] Inc. Unified EFI. Unified extensible firmware interface specification: Version 2.2d, November 2010.

[146] Ilja van Sprundel. Linux kernel bluetooth signed buffer index vulnerability. http://www.securityfocus.com/bid/12911.

[147] VMWare. VMWare, 2006. http://www.vmware.com.

[148] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, New York, NY, USA, 1993. ACM.

[149] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. volume 0, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[150] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *Proc. of the ACM workshop on Rapid malcode*, 2003.

[151] David A. Wheeler. SLOCCount, 2014.

[152] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proc. USENIX Symp. on Op. Sys. Design and Impl.*, Boston, MA, Dec 2002.

[153] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *Proc. Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, USA, October 2002.

[154] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proc. ACM SIGOPS Symp. on Op. Sys. Principles*, Brighton, UK, October 2005.

[155] C. Wright. Para-virtualization interfaces, 2006. http://lwn.net/Articles/194340.

[156] Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, New York, NY, USA, 2010. ACM.

[157] Jisoo Yang and Kang G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[158] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, January 2010.

[159] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: a retargetable framework for low-level inlined-reference monitors. In *Proceedings of the 22nd USENIX conference on Security*, SEC'13, pages 369–382, Berkeley, CA, USA, 2013. USENIX Association.

[160] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM.

[161] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, L. Szekeres, S. McCamant, D. Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573, 2013.

[162] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.

[163] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. Armor: Fully verified software fault isolation. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, pages 289–298, New York, NY, USA, 2011. ACM.

[164] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *USENIX Symposium on Operating System Deisgn and Implementation*, pages 45–60, Seattle, WA, USA, November 2006.